Programmation Fonctionnelle Avancée

Contrôle des connaissances 2010/2011 — 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
Barème définitif (total : **27** points)

Aucune réponse erronée, même avec des erreurs très graves, ne va réduire votre note : la contribution minimale d'une réponse à votre note finale est zéro. J'ai ajouté les commentaires en vert après la correction.

Première partie

Compréhension de code OCaml et typage (9 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*) ($\frac{1}{2}$ point) `1 :: 2` — Erreur : l'objet sur la droite n'est pas une liste Penser que `1 :: 2` soit une expression bien typée est une erreur **très grave** : probablement la majorité de ceux qui ont écrit ça n'ont jamais utilisé OCaml : en écrivant n'importe quelle fonction sur les listes on a besoin de travailler avec l'opérateur ou le motif *cons*, et on ne peut utiliser *cons* sans connaître au moins intuitivement ses règles de typage.

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

(*c*) ($\frac{1}{2}$ point) `(fun q -> []) (fun y -> y)  : 'a list` — Remarquez que le paramètre formel `q` n'est jamais utilisé ! Une fonction d'un paramètre qui retourne une constant, appelée avec un paramètre : le résultat, bien sûr, est la constante retournée. Mais presque personne n'a compris ça. Le fait que le paramètre soit une fonction aussi n'a aucune importance ! Les fonctions sont des objets comme tous les autres.

(*d*) ($\frac{1}{2}$ point) `fun x -> 3  : 'a -> int` Beaucoup de monde a pensé que le paramètre soit entier ; pourquoi ?

(*e*) ($\frac{1}{2}$ point) `fun x -> fun y -> x  : 'a -> 'b -> 'a` — Le deuxième paramètre est ignoré Beaucoup de monde a pensé que les paramètres soient entiers ; pourquoi ?

(*f*) ($\frac{1}{2}$ point) `fun x -> (x, x)  : 'a -> ('a * 'a)` Beaucoup de monde a pensé que le paramètre soit entier ; pourquoi ?

(*g*) ($\frac{1}{2}$ point) `let x = 1 in let y = "a" in [x, y]  : (int * string) list`

(*h*) ($\frac{1}{2}$ point) `let x = 1 in let x = "a" in x  : string` — La définition à l'intérieur "gagne"

(*i*) ($\frac{1}{2}$ point) `(1, "a")  : int * string`

(*j*) ($\frac{1}{2}$ point) `[1, "a"]  : (int * string) list`

(*k*) ($\frac{1}{2}$ point) `let x = 1 in if x < x then x else ()` —Erreur : la branche "then" a type `int`, la branche "else" a type `unit`

Q2. Écrire une expression dont le type soit :
Beaucoup d'étudiants ont écrit des définition globales au lieu des expressions, mais j'ai ignoré ça si la syntaxe de la définition était correcte, ou presque correcte. Écrire une expression (ou une définition) avec type différent d'un type fonction, en revanche, est une erreur **très grave**.

(*a*) ($\frac{1}{2}$ point) `bool -> int` Une fonction qui prend un entier comme paramètre et donne un booléen ; beaucoup de solutions possibles, peut-être la plus intuitive contient une conditionnelle : `fun b -> if b then 1 else 2` Plusieurs étudiants ont écrit des fonctions avec des motifs constants comme paramètres. Il s'agit de fonctions partielles, mais si le type est correcte je n'ai aucune objection. Exemple : `fun true -> 42`

(*b*) ($\frac{1}{2}$ point) `('a * 'b) -> 'a` Une fonction qui prend une couple et retourne un résultat avec le même type de l'élément sur la gauche... Pensez à `left` en TrivialML : `fun x -> match x with (left, _) -> left`

(*c*) ($\frac{1}{2}$ point) `'a -> ('a * int * ('a list))` Le paramètre est un objet de type arbitraire ; le résultat est une triple avec un objet du même type que le paramètre, un entier, et une *liste* d'objets du même type que le paramètre : `fun x -> (x, 42, [x])`

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*) ($\frac{1}{2}$ point) Pourquoi le message de warning est là ? Expliquez le problème. Le *pattern matching* n'est pas exhaustif : nous n'avons pas considéré le cas où une liste est vide et l'autre n'est pas vide.

(*b*) ($\frac{1}{2}$ point) Donnez un exemple d'application de `map2` démontrant le problème prévu dans le message de warning. Il faut donner deux listes avec tailles différents : `map2 (fun x y -> (x, y)) [1] [1; 2]`

(*c*) ($\frac{1}{2}$ point) Donnez une définition alternative de la fonctionne `map2` qui évite le message de warning. Vous allez avoir besoin d'une exception.
La version donnée traite déjà les cas où les tailles sont égales ; il faut juste ajouter un cas défaut :

```
exception ListsHaveDifferentLengths;;
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)
  | _, _ ->
    raise ListsHaveDifferentLengths;;
```

(*d*) ($\frac{1}{2}$ point)

```
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 ?
`foo` retourne une liste avec la moyenne des éléments correspondants des deux listes données : la moyenne des premiers éléments, la moyenne des deuxièmes, moyenne des troisièmes, etc... Tous les listes utilisées ici sont des listes des flottants :
`foo : (float list) -> (float list) -> (float list)`
J'ai demandé le type de `foo` : elle a *deux* paramètres. Beaucoup d'étudiants ont écrit des types avec quatre ou cinq flèches. Pas beaucoup de monde a articulé clairement l'idée des éléments avec index correspondent... J'ignore ça.

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, ...).

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*) (3 points) Donnez une règle de typage pour `foreach` :
e_1 doit être une liste du type correct (donc avec les éléments de type τ_1 ; x est utilisée dans le corps, donc il faut typer e_2 avec la prémisse $x : \tau_1$. Le corps, (et la boucle lui même aussi) est évalué pour ses effets bord, donc selon les conventions d'OCaml il a type `unit`.

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

Imposer $\Gamma, x : \tau_1 \vdash e_2 : \tau_2$ comme une prémisse et $\Gamma \vdash$ `foreach` $x : \tau_1$ `in` e_1 `do` $e_2 : \tau_2$ comme conclusion, avec le même τ_2 (sans la contraint $\tau_2 \equiv$ `unit`) est moins propre et moins cohérente avec la philosophie d'OCaml, mais acceptable. N'avoir pas compris qu'il faut écrire un type à la fin de la conclusion est **très grave**. Zéro points si vous n'avez pas imposé que e_1 soit une liste.

(*b*) (2 points) 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).
`e1` doit avoir type liste de `list_element_type`, et ce `list_element_type` doit être le même type déclaré pour `x` ; `e2` peut utiliser `x`, donc il faut étendre l'environnement avec le type de `x` pour typer `e2` ; le corps `e2` doit avoir type `unit`. Si tout ça est vrai, alors le type de la boucle entière est `unit` aussi :

```
    ...
    | EForEach(x, x_type, e1, e2) ->
      let e1_type = expression_type e1 rho gamma in
      (match e1_type with
       | TList(list_element_type) ->
           if x_type = list_element_type then
             let e2_type =
               expression_type e2 (Environment.bind x x_type rho) gamma in
             if e2_type = TUnit then
               TUnit
             else
               failwith "foreach: body doesn't have type unit"
           else
             failwith "wrong type declaration in foreach"
       | _ ->
         failwith "foreach: can only iterate on a list")
    ...
```

Votre code doit être cohérent avec votre règle !

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) (3 points) 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)

La liste grande est vivante pendant l'exécution de la boucle, et la boucle alloue de la mémoire : avec un garbage collector *tracing* comme *mark-sweep* ou *semispace* il faut visiter périodiquement le graphe des objets vivants, qui est très grand. Avec un conteur de références, par contre, la mémoire allouée pour chaque petite liste est libérée immédiatement à la fin de chaque itération, et aucune visite de la liste grande n'est jamais nécessaire. Donc dans le cas particulier de ce programme un conteur des références est plus efficace qu'un garbage collector.

Un compilateur optimisant «intelligent» pourrait mouvoir la création de la liste grande après la boucle, mais nous avons supposé un compilateur naïf. Je n'ai pas évalué négativement certaines argumentations absurdes : j'ai juste évalué positivement les réponses avec une référence, plus ou moins claire, à la nature synchrone des conteurs des références. J'ai été généreux.

(b) (2 points) 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 ?

La grande liste n'est jamais utilisée parce que la boucle est infinie, mais elle est toujours liée à la variable `x` : la grande liste est *garbage* sémantique, mais pas *garbage* syntaxique : elle n'est jamais détruite parce que les systèmes de gestion automatique de la mémoire (garbage collectors *tracing* et conteurs de références) ne peuvent reconnaître que le garbage syntaxique.

Un compilateur optimisant «intelligent» pourrait éviter du tout la création de la liste grande, mais nous avons supposé un compilateur naïf. Personne n'a utilisé les *keywords* «garbage syntaxique» et «garbage sémantique», mais beaucoup d'étudiants ont exprimé l'idée correcte, de façon plus ou moins claire. J'ai été généreux.

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*) (8 points) 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;;
```

Un entier signé peut être zéro, positif ou négatif : s'il n'est pas zéro, l'objet de type $\mathbb{N}^+$ à l'intérieur représente le *module* du nombre : le constructeur représente le signe.

```
module MakeInt (NatPlus : NatPlusInterface) : IntInterface = struct
  type t =
    | Zero
    | Positive of NatPlus.t
    | Negative of NatPlus.t;;

  type sign = ZeroSign | PositiveSign | NegativeSign;;

  let zero = Zero;;

  let sign i =
    match i with
      | Zero -> ZeroSign
      | Positive _ -> PositiveSign
      | Negative _ -> NegativeSign;;

  let predecessor i =
    match i with
      | Negative modulo ->
        Negative (NatPlus.successor modulo)
      | Zero ->
        Negative NatPlus.one
      | Positive modulo ->
        if NatPlus.is_one modulo then
          Zero
        else
          Positive (NatPlus.predecessor modulo);;

  let successor i =
    match i with
      | Negative modulo ->
        if NatPlus.is_one modulo then
          Zero
        else
          Negative (NatPlus.predecessor modulo)
      | Zero ->
        Positive NatPlus.one
      | Positive modulo ->
        Positive (NatPlus.successor modulo);;
end;;
```

Beaucoup de réponses sans aucune logique ; confusion entre signatures et modules, erreurs (graves) de syntaxe... La programmation ne peut pas être improvisée : il faut avoir utilisé le langage.
J'ai donné deux points (2 points) juste pour la syntaxe correcte (ou presque correcte), et une autre point (1 point) si la définition du module contient la partie banale (définition des types `t` et `sign`) : on devait juste copier, sans rien inventer. La définition de la fonction `sign` était banale aussi (j'avais donné la définition de `t` !), mais beaucoup d'étudiants ont simplement ignoré la signature `IntInterface`, en traitant le paramètre de `sign` comme un entier.
J'ai écrit explicitement que le type `NatPlus.t` est abstrait, et j'ai montré la signature `NatPlusInterface` aussi, où très

clairement le type `t` n'a aucune définition. Utiliser des objets de type `NatPlus.t` comme des entiers est une erreur **très grave** — et en fait vous ne connaissez pas la définition interne de `NatPlus.t` : il ne s'agit pas nécessairement d'un entier ! Voilà un exemple d'une implémentation possible de la signature `NatPlusInterface`, où les entiers ne sont pas utilisés du tout :

```
module NatPlus : NatPlusInterface = struct
  type t = One | Successor of t;;

  exception PredecessorOfOne;;

  let one = One;;

  let is_one n =
    match n with
      | One -> true
      | Successor _ -> false;;

  let successor n =
    Successor n;;

  let predecessor n =
    match n with
      | One -> raise PredecessorOfOne
      | Successor n_minus_one -> n_minus_one;;
end;;
```