Programmation – examen

Jeudi 18 janvier 2007, 9h-12h

Documents autorisés: version imprimée des transparents du cours, sujets et corrigés des TPs, notes de cours et de TPs.

1 Divers

1.1 Rangées

Dans ce qui a été vu en cours, la manipulation des enregistrements de Caml est très rigide. Un exemple standard d'utilisation est le suivant:

```
type enr = { a:int; b:string; mutable c:int }
let f x =
  begin
    x.c <- x.c*2;
    { a = x.a; b = "des chips"; c = x.c }
  end
```

et une erreur est engendrée si on définit `f` sans avoir auparavant défini `enr`, ou encore si on utilise un champ nommé `d` dans l'enregistrement renvoyé par `f`.

Dans cette partie, on se propose d'imaginer une version assouplie des enregistrements de Caml. Le principe est le suivant: le 'point' pour accéder aux champs sera remplacé par un `@`. Il n'y a plus besoin de définir explicitement des types d'enregistrement, et on travaille directement avec des enregistrements dont le type est de la forme { `c1:t1; c2:t2;...; ck:tk; ??`}, où les `ci` sont des noms de champs et les `ti` sont les types correspondants (comme pour les enregistrements 'classiques'), et **??** s'interprète comme *"... et potentiellement autre chose"*.

Par exemple, si on saisit la définition `let f x = x@a + x@b + 2`, on va obtenir le typage `f: {a:int; b:int; ??} -> int`.

1. Indiquez comment un type pourrait être inféré (et lequel) pour les trois définitions suivantes (ne vous contentez pas de donner le type inféré, mais montrez aussi comment cette information est synthétisée):

```
let f x = { a = x@a; b = x@b; c = "du chocolat"};;

let g y = if y@b then y@f1 32 else { t = y@f2 "miam"};;

let h t = t@x t@y;;
```

2. Proposer une définition de fonction qui engendre une erreur de typage due à une mauvaise utilisation de ces "enregistrements assouplis".

3. En se fondant sur l'idée du sous-typage tel qu'il a été discuté pour les modules de Caml, expliquer comment on pourrait introduire une notion de sous-typage entre types qui contiennent des **??**, et illustrer la chose à l'aide d'un exemple que vous commenterez.

1.2 Pair et impair

On considère la définition Caml suivante:

```
let rec pair n = n=0 || impair (n-1)
and impair n = n=1 || pair (n-1)
```

Un programmeur naïf, après avoir saisi ce qui précède, fait la définition

```
let pair n = (n mod 2) = 0
```

1. Si le programmeur espérait améliorer l'efficacité des fonctions `pair` et `impair`, force est de constater qu'il se fourvoya. Expliquez pourquoi.

2. Expliquer comment procéder (en réécrivant le code) de telle manière qu'en introduisant la première puis la seconde définition de `pair`, on obtienne *au final* des versions plus efficaces de `pair` et `impair`.

1.3 Listes de chaînes

Dans cette partie, on va exploiter l'idée du zipper, due à Gérard Huet. Il est par conséquent interdit, dans le code Caml que vous fournirez, d'utiliser un quelconque aspect impératif: pas le droit d'écrire `ref` *ni* `mutable`*.*

On va s'intéresser au stockage et à la modification de listes de chaînes de caractères. Ce qui en Caml se dit

```
type hist = string list
let l :  hist = ["http://bla.fr"; "http://bli.org"]
```

se prononce en C

```
struct hist
  {  char *contenu;
     struct hist *suivant;
  };

struct hist s1 = {"http://bli.org", NULL};
struct hist s2 = {"http://bla.fr", &s1};
```

1. On définit en Caml

   ```
   let tronque3 :  hist->hist = fun l ->
     let h,t = List.hd, List.tl in
     (h l)::(h (t l))::[h (t (t l))]
   ```

   (a) Définir la version 'C' de `tronque3` (sans trop se préoccuper des cas limite où la liste passée en argument est de taille inférieure ou égale à 2) – le rôle de cette fonction est uniquement de renvoyer la liste passée en argument de laquelle on a retiré tous les éléments à partir du quatrième.

   (b) Expliquer pourquoi, du point de vue de la gestion de la mémoire, le programmeur Caml est malheureux.

2. On adapte l'idée du zipper en définissant maintenant:

   ```
   type hist = (string list) * (string list)    (*   passé, présent::futur   *)
   ```

   On voit un élément de type `hist` comme une liste de sites web qu'on a visités avec un browser (ou *butineur*). Ainsi, cliquer sur *Back* sur le browser revient à appliquer la fonction définie par

   ```
   let back :  hist -> hist = function
   | (x::xs, ys) -> (xs, x::ys)
   | h -> h
   ```

   Donner par analogie les définitions de `forward` (aller vers l'avant dans la liste), `visit` (visiter un site dont on aura passé l'adresse en argument, comme si on saisissait cette adresse directement dans le browser), *en faisant attention à l'efficacité en mémoire.*

3. On revient à la représentation des listes en général, en oubliant l'exemple des browsers. Pour éviter les confusions, appelons `hist1` le type initial pour les listes (`type hist1 = string list`).

(a) Montrez comment on peut représenter les deux listes `["ha"; "ho"; "hu"; "he"]` et `["hi"; "hu"; "he"]` comme des éléments de type `hist1` de telle manière que la partie finale (`["hu"; "he"]`) soit partagée.

Donner un représentation correspondante dans le type `hist`.

(b) Illustrez similairement la représentation à l'aide du type `hist` de deux listes qui ont en commun une partie *initiale*.

4. Proposer la définition d'une fonction `modif : hist1 -> int -> string -> hist1` telle que l'appel `(modif h k s)` renvoie une liste contenant les mêmes éléments que `h` sauf le `k`ème (on commence à compter à zéro), qui est égal à `s`.

5. Comment utiliser l'idée du zipper pour programmer cela de manière plus efficace en mémoire? Donnez le code correspondant, et dites en quoi cette solution est particulièrement intéressante si, pour une raison ou pour une autre, on veut avoir accès aux deux versions de la liste initiale, *avant* et *après* la modification.

6. Expliquer (par exemple à l'aide de dessins illustrant ce qui se passe en mémoire) ce qui se passe du point de vue de la Garbage Collection lorsqu'on applique la fonction définie à la question précédente, pour le cas où l'on n'est pas intéressé à garder la version initiale.

2 Commentaire de texte

On s'intéresse au code qui est donné en dernière page de ce sujet d'examen. Lisez-le, les questions ci-dessous ont pour but de vous aider à le comprendre.

1. Donnez les types de `stocke`, `compress` et `ratatine`.

2. Montrez, en donnant des arguments aussi *rigoureux* que possible, que tout au long de l'exécution du code, la liste stockée en `memo` est triée suivant le premier argument des éléments qu'elle contient.

3. Que fait la fonction `stocke`?

   (expliquez comment la fonction effectue son calcul, ne vous contentez pas d'une ligne)

4. Que renvoie l'évaluation du code suivant?

```
let g l x = (x+3)::l in
List.fold_left g [] [49;29;8;6]
```

5. Que fait, au total, la fonction `ratatine`?

   (expliquez comment la fonction effectue son calcul, ne vous contentez pas d'une ligne)

6. Proposez du code OCaml, qui, lors de son évaluation, permettrait d'illustrer l'utilisation et l'intérêt de `ratatine`.

7. Supposons que le "`if x=e`" dans la fonction `search` est remplacé par "`if (egal x e)`".

   Dans une telle version plus générale de ce code, quel calcul pourrait effectuer la fonction `egal`? Quelle serait la signification du point de vue de ce que fait la fonction `ratatine`?

3 Régions

Dans cette partie, on s'intéresse à un mécanisme de gestion de la mémoire à l'exécution fondé sur l'idée de *région*[1].

[1] Pour les plus motivé(e)s d'entre vous, voir par exemple:
M. Tofte et J.-P. Talpin, *"Region-Based Memory Management"*, Information and Computation, 132(2): 109-176, 1997.

Lors de l'exécution des programmes, on renonce à l'organisation habituelle pile/tas, et on se contente d'une unique *pile de régions:* à chaque région correspond une zone mémoire, la libération des régions obéissant à la discipline caractéristique des piles, *last in, first out.*

Ainsi, à un moment donné de l'exécution d'un programme, les régions $\rho_1,\ldots,\rho_k$ sont actives, et on sait que la région ρ_{k-1} ne pourra être libérée qu'après la libération de ρ_k.

MLKit est une implémentation de Standard ML (un cousin de Caml, mais pour ce qui nous intéresse il n'y a pas de différence, et nous dirons simplement ML) utilisant les régions, et non un GC comme Caml. Lorsque l'utilisateur soumet un programme, celui-ci est traduit automatiquement vers un langage enrichi avec les constructions suivantes:

⟨*expr*⟩ **at** ρ
letregion ρ **in** ⟨*expr*⟩ **endregion**

ρ est une *variable de région*, et ⟨*expr*⟩ est une expression ML entrée par l'utilisateur.

◇ ⟨*expr*⟩ `at` ρ correspond à évaluer ⟨*expr*⟩, et stocker le résultat dans la région ρ.

◇ La construction `letregion` est une *définition locale de région:* dans l'exemple ci-dessus, on crée une nouvelle région ρ, on évalue ⟨*expr*⟩, et à la fin de ce calcul, la région ρ est libérée. Lors d'étapes ultérieures de la compilation, cette construction est mise en œuvre à l'aide d'instructions du type de `malloc` (pour le `letregion`) et `free` (pour le `endregion`).

À noter que le résultat de l'évaluation de ⟨*expr*⟩ devra être stocké dans une région vivante *"en amont de ρ",* i.e., existant avant que l'on n'entre dans le `letregion`.

1. Quels avantages présente ce principe par rapport à l'utilisation d'un Glaneur de Cellules?
2. Un système avec régions tel que MLKit privilégie certains "idiomes" de programmation. Ainsi, les transformations suivantes sont appliquées par le compilateur dès que possible:

   ◇ `let id1 =` ⟨*exp1*⟩ `and id2 =` ⟨*exp2*⟩ `in` ⟨*exp*⟩
   $\rightsquigarrow$ `let id2 = let id1 =` ⟨*exp1*⟩ `in` ⟨*exp2*⟩ `in` ⟨*exp*⟩ si `id1` n'est pas utilisé dans ⟨*exp*⟩;

   ◇ `let id1 =` ⟨*exp1*⟩ `in f` ⟨*exp*⟩ $\rightsquigarrow$ `f (let id1 =` ⟨*exp1*⟩ `in` ⟨*exp*⟩`)` si `id1`$\neq$`f`.

   Justifier l'utilité de ces transformations pour la programmation avec régions.

   Un exemple. Bien que MLKit implémente tout SML, *on ne s'intéressera ici qu'à un tout petit fragment de ML,* comportant les fonctions, les entiers, et les couples. On supposera que le système fait des allocations de manière systématique pour stocker toutes les données manipulées, y compris le code des fonctions. En revanche, les opérations de projection des couples `fst` et `snd`, ainsi que les opérations élémentaires sur les entiers (`+`,`*`, ...) sont des *primitives,* autrement dit elles ne sont pas implémentées comme des fonctions (ni stockées sur le tas).

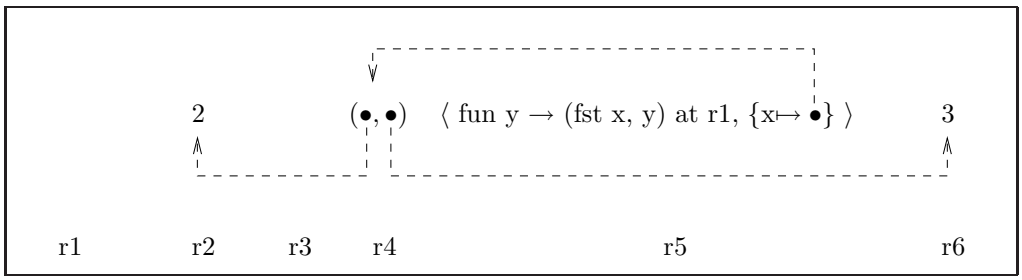
   Le programme tout simple suivant: `(let x = (2,3) in (fun y -> (fst x,y))) 5`

   sera traduit par le système en

```
letregion ρ4, ρ5 in
   ( letregion ρ6 in
     let x = (2 at ρ2, 3 at ρ6) at ρ4 in
       (fun y -> (fst x, y) at ρ1) at ρ5
   endregion )
   5 at ρ3
endregion
```

   On remarque en particulier que le stockage d'un couple fait a priori intervenir trois régions: une pour le "squelette" du couple (ici ρ_4), et une pour chaque composante (ici ρ_2 et ρ_6).

3. On veut dessiner l'évolution de la pile des régions lors de l'évaluation du code enrichi donné ci-dessus. Voici une représentation de la mémoire au moment où la fonction vient d'être stockée en mémoire:



Ci-dessus, on voit que les régions r1 et r3 sont vides, et la région r5 contient une *clôture* représentant la fonction: il y a l'argument y, le corps de la fonction, et une composante d'environnement qui indique à quoi est associée la variable x (qui est libre dans le corps de la fonction). On remarque aussi que les ρ_i ont été convertis en ri (mais c'est moins crucial).

Dessiner de même le moment juste avant l'application de la fonction, ainsi que l'état final.

4. Proposer et justifier une traduction pour l'expression suivante: `fst (5, (fun x -> 2))`.

5. Alors que l'insertion des `at` peut être faite de manière systématique (pour stocker entiers, couples et fonctions), l'insertion de `letregion...endregion` est plus complexe. Afin de garantir que les `letregion...endregion` sont placés de manière sensée dans le code, un système de types a été introduit, qui permet de typer les programmes du langage enrichi (avec les `at` et `letregion`).

   (a) Rappeler quelle propriété sur l'exécution des programmes garantit le système de types de Caml.

   (b) Similairement, que devrait garantir un système de types pour le langage enrichi?

   (c) Donner un exemple de programme du langage enrichi qui devrait être rejeté au moment du typage, en expliquant pourquoi.

6. Le système de types pour le langage enrichi associe à chaque expression un *couple*, dont la première composante est le type au sens usuel (comme en Caml) et la seconde composante est un *effet*, à savoir un ensemble de régions qui sont accédées par l'expression. Partant de cette idée, proposer un type et un effet pour le code enrichi suivant:

```
let x = 6 at ρ1 in
   (fun y ->  (x+y) at ρ3) at ρ2
```

Peut-on ici insérer un `letregion`? Comment pourrait s'exprimer la règle pour le typage d'un `letregion`?

7. Donnez des idées pour le typage "enrichi" des fonctions récursives. En particulier, comment traiter les fonctions récursives terminales?

```
let search e =
  let rec searchrec = function
  | [] -> raise Not_found
  | x::l -> if x=e then x else searchrec l
  in searchrec
;;

type info=int

type tree = Node of (info * tree list) ;;

type bucket = int * (tree list);;

let memo = ref ([]: bucket list);;


(* remarque: la doc de OCaml nous dit
  val List.rev_append : 'a list -> 'a list -> 'a list
  List.rev_append l1 l2 reverses l1 and catenates it to l2.
  This is equivalent to ((List.rev l1) @ l2), but rev_append is
           tail-recursive and more efficient.
*)

let stocke tree size =
  let enregistre (st,(n,l),r) =
    begin
      memo := List.rev_append st ((n,(tree::l))::r); tree
    end
  in
  let rec rem_rec st m = match m with
  | (((n,l) as p)::r) ->
      if n<size then rem_rec (p::st) r
      else if n=size then
        try search tree l with Not_found -> enregistre(st,p,r)
      else enregistre(st,(size,[]),m)
  | [] -> enregistre(st,(size,[]),m)
  in rem_rec [] !memo
;;

(* remarque: la doc de OCaml nous dit
val List.fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
List.fold_left f a [b1; ...; bn] is f (... (f (f a b1) b2) ...) bn.
*)

let rec compress = function
 | Node(i,arcs) ->
      let f (trees,size) t =
        let (t0,k) = compress t in
        ((t0::trees),size+k)
      in
      let (arcs0,s) = List.fold_left f ([],0) arcs
      in
      (stocke (Node(i,List.rev arcs0)) s,s+1)
;;

let ratatine tree = let (d,_) = compress tree in d;;
```