Examen. Session 1. Documents autorisés. Durée 1h30.

1 Structure de données *append list*

Soit le type suivant des "append list" :

```
type 'a app_list = Nil | Append of 'a app_list * 'a app_list | Cons of 'a * 'a app_list
```

La *append*-liste Cons (1, Append (Cons (2, Cons (3, Nil)), Cons (4, Nil))) représente la liste normale 1::([2;3] @[4]), où les *append* sont "en suspens" dans une structure de données.

Exercice 1 (*append list*)

1. Écrire les fonctions map et rev pour les *append list*.
2. Donner la complexité de la fonction rev en fonction du nombre d'éléments, en justifiant votre réponse.
3. Écrire la fonction fold_right pour les *append list*.

Solution 1

```
let rec map f = function
| Nil            -> Nil
| Cons (t, q)    -> Cons (f t, map f q)
| Append (l, r)  -> Append (map f l, map f r)

let rec rev = function
| Nil            -> Nil
| Cons (t, q)    -> Append (rev q, Cons (t, Nil))
| Append (l, r)  -> Append (rev r, rev l)

let rec fold_right f l e =
match l with
| Nil            -> e
| Cons (t, q)    -> f t (fold_right f q e)
| Append (l, r)  -> fold_right f l (fold_right f r e)
```

Soit la fonction de conversion naïve suivante de type 'a app_list -> 'a list :

```
let rec to_list al =
  match al with
  | Nil                -> []
  | Cons (t, q)        -> t::(to_list q)
  | Append (al1, al2)  -> (to_list al1) @ (to_list al2)
```

Exercice 2 (conversion)

1. Donner la complexité du pire cas de to_list en fonction du nombre d'éléments en justifiant votre réponse.
2. En utilisant un paramètre accumulateur supplémentaire, obtenir une nouvelle fonction de conversion, de complexité linéaire en fonction du nombre d'éléments.

Solution 2

```
(* pire cas de to_list : app_list est un peigne à gauche, donc O(n*n) *)
let rec to_list_aux al acc =
match al with
| Nil            -> acc
| Cons (t, q)    -> t::(to_list_aux q acc)
| Append (l, r)  -> to_list_aux l (to_list_aux r acc)
```

2 Continuations

On utilisera les primitives standard reset et shift, ou alternativement new_prompt, push_prompt et shift de la librairie Delimcc, ainsi que le type :

```
type ('reset, 'shift) res = Done of 'reset | Request of 'shift -> ('reset, 'shift) res
```

Exercice 3 (compréhension)

Déterminer les types des définitions x1, x2 et x3 suivantes :

1. `let x1 = reset (fun () -> Done (if shift (fun k -> Request k) then "hello"else "hi") ^ "world")`
2. `let x2 = reset (fun () -> Done (let x = shift (fun k -> Request k) in x * x))`
3. `let x3 b = reset (fun () -> Done (5 * (if b then (shift (fun k -> Request k) else 0) + 3 * 4)))`

Déterminer les résultats produits par l'évaluation des expressions suivantes :

1. `match x1 with | Done i -> i | Request k -> k true`
2. `match x2 with | Done i -> i | Request k -> 0`
3. `match x3 false with | Done i -> i | Request k -> k 6`

Solution 3

```
x1 : (string, bool) res; x2 : (int, int) res; x3 : bool -> (int, int) res
match x1 ... -> "hello world"; match x2 ... -> 0; match x3 false ... -> 12
```

On s'intéresse à l'implantation d'un mécanisme d'émission/réception de messages entre processus coopératifs, à travers un médium unique. On dispose d'un nombre fixé de processus, orchestrés par un scheduler. Chaque processus qui s'exécute rend la main au scheduler uniquement soit en terminant, soit en postant un message sur le médium (émission non bloquante), soit en attendant qu'un message soit disponible (réception bloquante). Aucun ordre particulier n'est attendu en ce qui concerne les émissions et réceptions. Pour simplifier, les messages seront des nombres entiers.

Le scheduler gère les messages dans le medium, les processus bloqués en réception, les autres processus inactifs, ainsi que le processus courant. Il réveille prioritairement les processus récepteurs tant qu'il reste une valeur à consommer dans le medium, avant d'envisager l'exécution d'un autre processus inactif.

Exercice 4 (messages)

1. Définir un nouveau type res en fonction des interactions spécifiées entre processus et scheduler.
2. Compléter le `type proc = unit -> ... res` corespondant au type des processus à scheduler.
3. Définir les primitives `send : int -> unit`, `receive : unit -> int`.
4. Définir la fonction `scheduler : proc list -> unit`, qui va scheduler une liste initiale de processus, dans un ordre quelconque.

Solution 4

```
type res =
| Done
| Send of int * (unit -> res)
| Receive of (int -> res);;

let p = Delimcc.new_prompt ();;

let send v = Delimcc.shift p (fun k -> Send (v, k));;
let receive () = Delimcc.shift p (fun k -> Receive k);;

let scheduler procs =
  let rec loop result others rcvs medium =
  match result with
```

```
  | Done          -> comm others rcvs medium
  | Send (v, k)  -> comm (k::others) rcvs (medium@[v])
  | Receive k    -> comm others (rcvs@[k]) medium
  and comm others rcvs medium =
  match rcvs, medium, others with
  | kr::qr, vm::qm, _     -> loop (kr vm) others qr   qm
  | _     , _     , o::qo -> loop (o ())  qo       rcvs medium
  | _     , _     , []    -> Done
  in ignore (Delimcc.push_prompt p (fun () -> loop Done procs [] []));;
```

3 Typage avançé

On s'intéresse à l'introduction de paramètres supplémentaires dans le type `list`, afin de spécifier le comportement de certaines fonctions. On aura besoin des types `zero` et `_ succ` pour représenter des entiers naturels, du type `nil` pour indiquer la fin d'une liste, et enfin du type `'a list` :

```
type zero = private Dummy1
type _ succ = private Dummy2
type nil = private Dummy3
type 'a list = Nil | Cons of 'a * 'a list
```

On s'intéresse d'abord aux n-listes, où l'on introduit un paramètre supplémentaire indiquant la taille de la liste.

Exercice 5 (n-listes)

1. Définir le type `type ('a, 'n) nlist` représentant les listes d'éléments de type `'a` et de taille `'n`. On reprendra les mêmes contructeurs `Nil` et `Cons`.
2. Définir la fonction `map : ('a -> 'b) -> ('a, 'n) nlist -> ('b, 'n) nlist` sur le modèle de `List.map`.
3. Donner le type et définir la fonction `snoc` qui ajoute un élément à la fin d'une n-liste.
4. Donner le type et définir la fonction `tail` qui prend une n-liste non vide et renvoie la queue de cette n-liste. Cette fonction ne doit pas lever d'erreurs à l'exécution.
5. Définir la fonction `rev : ('a, 'n) nlist -> ('a, 'n) nlist` sur le modèle de `List.rev`.

On s'intéresse maintenant aux listes hétérogènes, ou h-listes. Une h-liste accepte des éléments de tout type. Le paramètre de type représente la suite des types (potentiellement différents) de tous les éléments de la h-liste. Cette suite est écrite sous la forme d'un produit de types, terminé par `nil`. Par exemple, on aurait le typage suivant : `Cons (1, Cons (true, Nil)) : (int * (bool * nil)) hlist`

Exercice 6 (h-listes)

1. Donner le type attendu de : `Nil`.
2. Donner le type attendu de : `Cons (true, Cons (1, Cons (false, Nil)))`.
3. Définir le type `'p hlist` représentant les listes hétérogènes.
4. Donner le type et définir la fonction `tail` qui prend une h-liste (non vide) et renvoie la queue de cette h-liste. Cette fonction ne doit pas lever d'erreurs à l'exécution.
5. Donner le type et définir la fonction `add` qui prend deux entiers aux deux premières positions d'une h-liste et insère leur somme en tête de la queue (la h-liste privée des deux premiers entiers). Cette fonction ne doit pas lever d'erreurs à l'exécution.