Examen Final

IFT-2035

June 11, 2015

Directives

- Documentation autorisée: aucune.
- Répondre sur le questionnaire dans l'espace libre qui suit chaque question. Utiliser le verso de la page si nécessaire.
- Chaque question vaut 5 points pour un total maximum de 25 points.
- Les questions ne sont pas placées par ordre de difficulté.

0 Nom et prénom (1 point de bonus)

Écrire son nom et prénom et son code permanent en haut de chaque page.

1 Fonctions d'ordre supérieur

Ré-écrire les fonctions Haskell ci-dessous sans utiliser d'appel récursif, en utilisant *foldr* à la place:

$$
\begin{aligned}
&map\ f\ [] = [] \\
&map\ f\ (x : xs) = f\ x : map\ f\ xs
\end{aligned}
$$

$$
\begin{aligned}
&sum\ [] = 0 \\
&sum\ (n : ns) = n + sum\ ns
\end{aligned}
$$

$$
\begin{aligned}
&translate\ \_\ [] = [] \\
&translate\ (xo, yo)\ ((x, y) : vs) = (x + xo, y + yo) : translate\ (xo, yo)\ vs
\end{aligned}
$$

$$
\begin{aligned}
&append\ []\ ys = ys \\
&append\ (x : xs)\ ys = x : append\ xs\ ys
\end{aligned}
$$

$$
\begin{aligned}
&length\ [] = 0 \\
&length\ (\_ : xs) = length\ xs + 1
\end{aligned}
$$

Au cas où votre mémoire flanche, *foldr* peut être défini comme suit:

$$
\begin{aligned}
&foldr :: (\alpha \to \beta \to \beta) \to \beta \to [\alpha] \to \beta \\
&foldr\ f\ i\ [] = i \\
&foldr\ f\ i\ (x : xs) = f\ x\ (foldr\ f\ i\ xs)
\end{aligned}
$$

2 Gestion mémoire

Soit le code C suivant:

```
typedef struct list list;
struct list {
  int n;
  void *elem;
  list *next;
}

list *list_cons (void *elem, list *tail)
{ list *l = malloc (sizeof (list));
  l->n = 1;
  l->elem = elem;
  l->next = tail;
  return l;
}

void *list_head (list *l)
{ return l->elem; }

list *list_tail (list *l)
{ return l->next; }

list *list_copy (list *l)
{ l->n++; return l; }

list *list_free (list *l)
{ l->n--;
  if (l->n == 0) {
    list_free (l->next);
    free (l);
} }
```

1. Quel style de gestion mémoire est-il utilisé par cette librairie? Justifier.
2. Décrire les conditions que ce code est censé préserver pour s'assurer que la gestion de la mémoire est correcte.
3. Il y a un bug dans le code ci-dessus, qui a à voir avec la gestion de la mémoire; lequel? Quel genre de problème peut-il engendrer? Corriger le bug.
4. Si on essaie de manipuler des listes de listes, on se heurte à une limitation du code ci-dessus. Laquelle? Comment peut-on la corriger?

3 Raisonner

Soit le morceau de code suivant, où `f` est une fonction quelconque que l'on ne connaît pas (e.g. définie dans un autre fichier pas encore écrit) et où le langage utilise une syntaxe de style C, et une librairie similaire à celle du TP2:

```
void foo (void (*f) (String*, int)) {
  int size = 5;
  int tmp = 0;
  String *s1 = str_alloc (1);
  String *s2 = str_alloc (size);
  char *cs = str_data (s1);
  cs[0] = 'a';
  f (s2, size);
}
```

On aimerait savoir (par exemple pour décider de la validité de certaines optimisations) si certaines conditions sont nécessairement toujours vraies à la fin du bloc ci-dessus (i.e. quand l'exécution de `f` se termine). On s'intéresse plus particulièrement aux conditions suivantes:

- `str_size (s1) == 1`
- `str_data (s1) == cs`
- `str_data (s1)[0] == 'a'`
- `size == 5`
- `tmp == 0`

Dans chacun des cas suivants, indiquer lesquelles de ces 5 conditions sont nécessairement vraies et si non, justifier pourquoi pas:

1. Le langage est exactement comme C: portée statique, passage d'arguments par valeur, affectation autorisée.
2. Le langage est comme C sauf qu'une fois initialisées, les *variables* sont immuables.
3. Le langage est comme C sauf que les arguments sont passés par référence.
4. Le langage est comme C mais avec portée dynamique.

4 Prolog pur

Soit la règle Prolog de concaténation de liste:

```
append([], B, B).
append([X | A], B, [X | C]) :- append(A, B, C).
```

Attention à bien distinguer les , des |: `[A, B]` = `[A | [B | []]]`

1. Dessiner les arbres de preuves importants de la requête:

   ```
   append([X, 5], X, [X, 5, 10]).
   ```

2. Écrire une régle `sublist(A,B,C)` qui dit que C est une sous-liste de A et de B. Par exemple `[1, 2]` est une sous-liste de `[3, 1, 2, 4]` (mais pas de `[3, [1, 2], 4]`).

3. Écrire une régle `repeat(A,B)` qui dit que B est une sous-liste de A qui apparait au moins deux fois dans A.

4. Écrire une régle `exact(A,B)` qui dit que B est une sous-liste de A qui apparait exactement deux fois, ni plus ni moins, dans A.

5 Macros

Soit la macro until similaire à while mais avec la condition inversée:

```
(define-macro (until cond body)
  `(letrec ((loop (lambda ()
                    (if ,cond
                        ,cond
                      (begin ,body (loop))))))
     (loop)))
```

Done `(until COND EXP)` est similaire à `(while (not COND) EXP)` à la différence que until renvoie aussi la valeur finale de `COND`.

1. Montrer un usage de cette macro qui souffre de l'usage de l'appel par nom.
2. Corriger le code pour qu'il utilise l'appel par valeur à la place.
3. Montrer un usage de votre macro corrigée qui souffre de capture de nom.
4. Corriger alors votre macro, mais sans utiliser gensym.

6 Types, le retour

Dans le code ci-dessous, $\bullet$ représente une expression manquante. Donner le type de l'expression manquante. E.g. pour la question 0, la réponse pourrait être: $\bullet : \mathsf{Int} \to \alpha$.

0. $\bullet\ 1$
1. $[\bullet, 2]$
2. $(\lambda x \to 13 * \bullet)\ []$
3. $[\bullet, [1, 2, 3]]$
4. $(\text{"Haskell"}, (1, 2), \bullet)$
5. $\lambda x \to \lambda y \to \bullet\ x$
6. $\mathsf{let}\ x = \bullet\ \mathsf{in}\ map\ x\ (x\ 1)$
7. $\lambda x \to \bullet\ 4\ x$
8. $(\bullet\ (snd\ map))$
9. $map\ \bullet\ [fst, snd]$
10. $map\ fst\ [\bullet]$

Rappel: les fonctions *map* et *fst* sont (pré)définies comme suit:

$$
\begin{aligned}
&map\ f\ [] = [] \\
&map\ f\ (x : xs) = f\ x : map\ f\ xs
\end{aligned}
$$

$$
\begin{aligned}
&fst\ (x, y) = x \\
&snd\ (x, y) = x
\end{aligned}
$$

Et $x + y$ n'est rien de plus que du sucre syntaxique pour $(+)\ x\ y$.