Examen Final

IFT-2035

June 11, 2015

Directives

- Documentation autorisée: une page recto de notes manuscrites.
- Répondre sur le questionnaire dans l'espace libre qui suit chaque question. Utiliser le verso des pages 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 Macros

Soit la macro ccase similaire au case du TP1:

```
(define-macro (ccase x branch-n branch-c)
  (assert (eq 'nil  (car branch-n)))
  (assert (eq 'cons (car (car branch-c))))
  (let* ((exp-n   (car (cdr branch-n)))
         (exp-c   (car (cdr branch-c)))
         (pat-c   (cdr (car branch-c)))
         (var-car (car pat-c))
         (var-cdr (car (cdr pat-c))))
    `(if (null? ,x)
         ,exp-n
       (let ((,var-car (car ,x))
             (,var-cdr (cdr ,x)))
         ,exp-c))))
```

Donc l'expansion de `(ccase p (nil 13) ((cons x y) (+ x y)))`
est `(if (null? p) 13 (let ((x (car p)) (y (cdr p))) (+ x y)))`.

1. Cette définition de la macro souffre d'un problème. Dire quel genre de problème, et montrer un usage de cette macro qui souffre de ce problème.
2. Indiquer en détail comment changer le code pour corriger ce problème.
3. Cette correction introduit généralement un autre problème. Dire quel genre de problème, et montrer un usage de cette macro qui en souffre.
4. Indiquer en détail comment changer le code pour corriger ce problème.

2 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. $\lambda x \to (\bullet + 13 * x)$
3. $[\bullet, [1, 2, 3]]$
4. $([1, 2], \bullet)$
5. $map\ \bullet\ [5, 6, 7]$
6. $map\ snd\ [\bullet]$
7. $\lambda x \to (x + \bullet\ x)$
8. $\lambda x \to \lambda y \to (x\ y + \bullet\ x)$
9. $\mathsf{let}\ x = \bullet\ \mathsf{in}\ map\ x\ (x\ [42])$
10. $\lambda x \to (\bullet\ 13\ 4, x)$

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

$$map\ f\ [] = []$$
$$map\ f\ (x : xs) = f\ x : map\ f\ xs$$
$$snd\ (x, y) = y$$

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

3 Gestion mémoire

Soit les déclarations suivantes d'une libraire de gestion mémoire en C:

```
struct holder* holder_new  (void);
void*          holder_alloc (struct holder* h, int bytes);
void           holder_free  (struct holder* h);
```

1. Quel style de gestion mémoire est offert par cette librairie? Justifier.
2. Décrire les conditions que les utilisateurs de cette librairie doivent obéir pour que le programme ne souffre pas de problème de gestion mémoire.
3. Indiquer les deux types différents de problèmes qui peuvent apparaître si ces conditions ne sont pas obéies.
4. Indiquer les avantages de cette librairie par rapport au malloc/free classique.

4 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) (char*, int)) {
  int x = 3, y = 2, z = 1;
  char *s1 = "haskell";
  char *s2 = strcpy (s1);      /* fait une copie de s1 */
  char *s3 = s2 + 5;
  while (--x > 0) {
    int x = 2 * y;
    f (s3, y);
} }
```

On aimerait savoir (par exemple pour décider de la validité de certaines optimisations) si certaines conditions sont nécessairement toujours vraies à la fin de la fonction ci-dessus (i.e. quand l'exécution du `while` se termine).
On s'intéresse plus particulièrement aux 6 conditions suivantes:

```
s2[5] == 'l'          x == 0
s2[1] == 'a'          z == 1
s1[1] == 'a'          y == 2
```

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 que les arguments sont passés par référence.

3. Le langage est comme C mais avec portée dynamique.

3 Objets géométriques

Soit le langage hypothétique Claskell qui combine Haskell avec un système orienté objet à base de classes traditionnelles.

```
class Drawable { method draw :: Image → Image; };
class Square extends Drawable
{ size :: Float; method draw(d :: Image) {...}; };
class Circle extends Drawable
{ radius :: Float; method draw(d :: Image) {...}; };
```

Sachant donc que *Circle* $\subseteq$ *Drawable* et *Square* $\subseteq$ *Drawable*, indiquer quelles relations de sous-typage seraient valides:

1. *Square* $\subseteq$ *Square*
2. *Circle* $\to$ *Circle* $\subseteq$ *Circle* $\to$ *Drawable*
3. *Circle* $\to$ *Circle* $\subseteq$ *Drawable* $\to$ *Drawable*
4. List *Square* $\subseteq$ List *Drawable*
5. (*Square* $\to$ *Drawable*) $\to$ *Circle* $\subseteq$ (*Square* $\to$ *Circle*) $\to$ *Drawable*

Soit un objet c :: *Circle* et soit une fonction *zoom* qui prend un facteur d'agrandissement et un objet de la classe *Drawable*, et renvoie un objet équivalent mais agrandi. Donner le type de:

6. *zoom*
7. *zoom* 2.0 c

Finalement, un programmeur tout aussi hypothétique décide de traduire le code vers Haskell (et ses classes de type):

```
class Drawable a with draw :: Image → Image
data Square = Square { size :: Float }
data Circle = Circle { radius :: Float }
instance Drawable Square with
  draw d = ...
instance Drawable Circle with
  draw d = ...
c :: Circle = ...
```

Donner le type Haskell qui en résulte pour:

6. *zoom*
7. *zoom* 2.0 c