Examen de Compilation Avancée – M2 Informatique 2016-2017

Durée: 2h30

Notes de cours autorisées.

Nota :

- Dans les réponses à écrire en syntaxe FOPIX ou KONTIX, vous pourrez utiliser la notation $[|a; \dots; c|]$ inspirée d'OCAML pour la création et le remplissage d'un tableau.
- Pour les réponses à donner en JAVIX, vous pourrez ignorer les Box/Unbox.

Exercice 1 (*Fibonacci Forever*) Voici une définition possible de la fonction fibonacci en OCAML :

```
let rec f n = if n < 2 then 1 else f (n-1) + f (n-2)
let fib7 = f 7
```

1. Traduisez ce programme en JAVIX.
2. Donner en OCAML une version récursive terminale de fibonacci, à base d'accumulateur(s). Comment calcule-t-on fib7 avec votre code ? Outre l'aspect terminal, quel est l'autre avantage de votre version par rapport au f donné ci-dessus ?
3. Traduisez votre réponse précédente en JAVIX, en optimisant les appels terminaux. Quelle différence reste-t-il avec la compilation d'un fibonacci impératif ?
4. Utiliser l'algorithme de mise en CPS vu en cours pour convertir le programme f de départ en KONTIX. Comparer cette version de fibonacci et les précédentes.
5. On considère maintenant la fonction suivante :

```
let rec g n = if n = 0 then 0 else n - g (g (n-1))
```

Utiliser l'algorithme de mise en CPS vu en cours pour convertir g en KONTIX.

6. Au fait, quel est le résultat de g sur fib7 ? Conjecturer l'action de g sur un nombre de fibonacci différent de 1. Conjecturer l'action de g sur une somme de nombres de fibonacci distincts et différents de 1.

□

Exercice 2 (*Taille de pile en JAKIX*) A partir d'un programme KONTIX :

1. Donner l'algorithme permettant de calculer la taille maximale de pile qui sera nécessaire au programme JAKIX correspondant. Attention lors des appels de fonctions : la mise en place des arguments doit se faire progressivement !
2. Donner les invariants qui doivent être respectés en entrée et sortie de blocs par le programme JAKIX.
3. Donner la taille maximale de pile nécessaire pour le programme KONTIX de la question 1.4.

□

Exercice 3 (Trampoline) On considère la fonction `rev_append` suivante :

```
let rec rev_append l1 l2 = match l1 with
| [] -> l2
| x::l1' -> rev_append l1' (x::l2)
```

1. Utilisez la technique du trampoline pour transformer cette fonction récursive en une fonction OCAML non-récursive utilisant une boucle `while`.
2. Y avait-il ici une solution plus directe pour obtenir un `rev_append` via une boucle `while` ? Si oui laquelle ?
3. Dérécursifiez le code suivant à l'aide d'un trampoline :

```
type tree = Node of int * tree list

let rec sum_tree s ll (Node (n,l)) =
  sum_treelist (n+s) ll l

and sum_treelist s ll = function
| [] -> sum_treelistlist s ll
| t::l -> sum_tree s (l::ll) t

and sum_treelistlist s = function
| [] -> s
| l::ll -> sum_treelist s ll l

let sum t = sum_tree 0 [] t

let ex = sum (Node (3, [Node (4, []);
                      Node (5, [Node (7, [])])]))
```

□

Exercice 4 (CPS et typage) En OCAML, comment peut-on profiter du typage pour s'assurer qu'une fonction récursive écrite en style CPS ne va jamais rendre la main autrement qu'en lançant la continuation qu'elle a reçu ? Lors d'une mise en CPS d'une fonction, quelle transformation subit son type ? □