Intra - ift2035 Nom:_____ Matricule:_____

Examen Intra

IFT-2035

May 18, 2021

Directives

- Répondre dans le *fichier de réponses* associé.
- Chaque question vaut 3 points, pour un total de 15 points.
- Les questions ne sont pas placées par ordre de difficulté.

0 Identification et code d'honneur

1. Écrire l'identification de votre groupe.
2. Écrire pour chaque membre du groupe que vous promettez de faire cet examen sans faire recours à de l'aide extérieure.

1 Syntaxe

Soient les expressions suivantes en notation postfixe:

1. `a x y d * + -`
2. `x d - x d - -`
3. `a b c + /\ d /\`
4. `a b c * sin + d sin > not`
5. `a b = not c d - d sqrt e * < || f &&`
6. `x 0 < nil z a b + cons if`

Après s'être souvenu que la mise à la puissance (représentée par `/\` ci-dessus) est associative à droite, écrire ces expressions en format infixe, préfixe, et en format parenthèsé à la Psil/Lisp. En notation infixe utiliser un *minimum* de parenthèses.

2 Types

Donner le type des expressions Haskell ci-dessous, mais pour une version simplifiée de Haskell qui n'a pas de classes de types et où les nombres sont tous des entiers de type Int.
Par exemple, la réponse pour $(2, 3)$ serait: (Int, Int)
Le type donné devrait être aussi polymorphe que possible.

1. [("Emacs", 4), ("Typer", 6)]
2. $\lambda x \to x\ (+)$
3. $\mathsf{let}\ f = \lambda f \to f + 1\ \mathsf{in}\ f + 2$

Donner le type des trous $\bullet$ suivants.
Par exemple, la réponse pour $\bullet\ +\ 1$ serait: Int

4. $map\ \bullet\ [snd, fst]$
5. $map\ snd\ \bullet$
6. $\mathsf{let}\ f = \bullet\ \mathsf{in}\ snd\ f + fst\ f$

Donner un exemple de code qui a le type demandé.
Par exemple, la réponse pour $\alpha \to \alpha$ serait: $\lambda x \to x$

7. $\mathsf{Int} \to (\mathsf{Int} \to \mathsf{Int})$
8. $(\alpha \to \beta) \to \alpha \to [\beta]$
9. $[(\mathsf{Int} \to \mathsf{Int}, \mathsf{Int})]$

Précisions: Les fonctions *map*, *fst*, et *snd* sont (pré)définies comme suit:

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

$fst\ (x, y) = x$
$snd\ (x, y) = y$

3 Portée

Soit le code ci-dessous qui est écrit en Haskell et utilise donc la portée lexicale:

```
λa◯-> λa◯->
   let b◯ c◯ = λd◯-> c◯ + a◯ in
   let d◯ = λc◯-> a◯ (b◯ c◯) in
   let a◯ (b◯, c◯) = d◯ (b◯, d◯)
   in λc◯-> a◯ (b◯, c◯)
```

Renommer toutes les variables (e.g. en y ajoutant un 0, 1, 2, ... dans le cercle) pour que chaque variable ait un nom différent des autres. Bien sûr ce renommage ne doit pas changer la sémantique du code.
Écrire juste les numéros ajoutés (dans l'ordre!) dans le fichier réponses.

4 Évaluateur

Soit l'évaluateur ci-dessous, écrit en Haskell, pour un langage fonctionnel proche du λ-calcul:

```
type Var = ...                  type Env = ...
data Val = Vnum Int             data Exp = Enum Int
         | Vfun Var Exp                  | Evar Var
                                         | Efun Var Exp
                                         | Ecall Exp Exp
                                         | Elet Var Exp Exp
env_lookup :: Env -> Var -> Val
env_add :: Env -> Var -> Val -> Env

eval env (Enum n) = n
eval env (Evar x) = env_lookup x env
eval env (Ecall fun actual)
  = case fun of
      Vnum n -> error "Un nombre n'est pas une fonction"
      Vfun formal body ->
        eval (env_add env formal (eval actual)) body
```

Où *env_lookup* et *env_add* sont des fonctions qu'on présume prédéfinies, qui permettent respectivement de trouver les infos d'une variable et d'ajouter une variable dans un environnement.

1. Haskell signale des erreurs de typage: Indiquer comment corriger le code.
2. *eval* implémente-t-il l'appel par nom (CBN) ou l'appel par valeur (CBV)? Justifier.
3. *eval* implémente-t-il la portée *lexicale* ou la portée *dynamique*? Justifier.
4. Indiquer comment changer le code pour obtenir l'autre sorte de portée.
5. Haskell se plaint qu'*eval* n'est pas "exhaustive": Compléter sa définition.

5 Structures de données fonctionnelles

Soit le type suivant en Haskell qui définit un arbre binaire que l'on peut utiliser pour représenter une table associative (qui associe des *clés* de type [Bool] à des valeurs de type quelconque `b`):

```
data Maybe a   = Nothing | Just a
data TreeMap b = Empty | Node (Maybe b) (TreeMap b) (TreeMap b)
```

Par exemple, l'arbre qui associe 42 à `[]`, 17 à `[True]`, et 23 à `[False]` aurait la forme:

```
Node (Just 42) (Node (Just 17) Empty Empty)
               (Node (Just 23) Empty Empty)
```

L'opération de recherche *tmLookup* a la forme suivante:

```
tmLookup [] (Node v _ _ ) = v
...
```

et fonctionne sur le principe de lire la liste de booléens comme une séquence de commandes où *True* indique de descendre dans le sous-arbre de gauche et *False* indique de descendre dans le sous-arbre de droite.

1. Donner le type de la fonction *tmLookup* ci-dessus.
2. Compléter la fonction *tmLookup* ci-dessus.
3. Donner des type raisonnables pour les fonctions *tmInsert* et *tmRemove* qui permettent respectivement d'insérer et d'enlever un élément dans une table.
4. Indiquer l'ordre de grandeur du nombre d'opérations nécessaires en temps CPU et en allocations mémoire pour exécuter *tmInsert* ou *tmRemove* en fonction du nombre N de valeurs dans la table et de la longueur M de la clé.