Série d'exercices #3

IFT-2035

May 12, 2021

3.1 Micro optimizer

Soit le code Haskell ci-dessous:

```
data Exp = Enum Int          -- Une constante
         | Eplus Exp Exp     -- e1 + e2
         | Etimes Exp Exp    -- e1 * e2

optimize ::  Exp -> Exp
```

Exp est un type qui représente des expressions simples incluant uniquement des opérateurs arithmétiques. Écrire la fonction *optimize* qui va essayer de simplifier une expression en éliminant *toutes* les multiplications par 1 et 0, ainsi que les additions à 0.

3.2 Des fonctions pour table

Définir en Haskell des fonctions pour gérer des tables associatives sans utiliser de constructeur (tels que `(:)` ou des types algébriques). Plus précisément, définir:

`empty = λx -> error "Not found"`	Une table vide
`add x v t`	Ajouter un lien `x` $\mapsto$ `v` à la table `t`
`lookup t x`	Renvoie la valeur `v` liée à `x` dans `t`

De telle manière que:

```
t = add "x" 1 (add "y" 2 empty)
lookup t "y" ⟹ 2
lookup t "z" ⟹ <error>
```

Vu que les constructeurs de données ne peuvent pas être utilisés, les tables seront nécessairement représentées par des fermetures (i.e. il faudra utiliser des fonctions d'ordre supérieur).

3.3 Des trous typés

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: $\bullet : \mathsf{Int} \to \alpha$. Comme d'habitude, vous pouvez présumer que toutes les entités numériques sont de type Int.

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

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

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

3.4 Mini évaluateur

Soit les déclarations de type suivantes utilisées pour un mini-interpréteur d'expressions arithmétiques:

```
type Var = String

-- Expressions du code source en forme ASA.
data Exp = Enum Int               -- Une constante
         | Evar Var               -- Une variable
         | Elet Var Exp Exp       -- Une expr "let x = e1 in e2"
         | Ecall Exp Exp          -- Un appel de fonction

-- Valeurs renvoyées.
data Val = Vnum Int               -- Un nombre entier
         | Vprim (Val -> Val)     -- Une primitive
```

Les fonctions prédéfinies sont l'addition, la soustraction, la multiplication, et la division, liées aux variables "+", "-", "*", et "/", respectivement. Ces fonctions prennent deux arguments qui sont passés de manière currifiée. Par exemple une expression telle que "let $x = 3$ in $x + 4$" est représentée par la structure suivante de type *Exp*:

```
sampleExp = Elet "x" (Enum 3)
                 (Ecall (Ecall (Evar "+") (Evar "x")) (Enum 4))
```

L'environnement initial prédéfini les quatre fonctions:

```
mkPrim :: (Int -> Int -> Int) -> Val
mkPrim f = Vprim (\(Vnum x) -> Vprim (\(Vnum y) -> Vnum (f x y)))

-- L'environnement initial qui contient toutes les primitives.
type Env = [(Var, Val)]
pervasive :: Env
pervasive = [("+", mkPrim (+)), ("-", mkPrim (-)),
             ("*", mkPrim (*)), ("/", mkPrim div)]
```

Écrire la fonction *eval* qui prend un environnement qui décrit les variables liées (et leur valeur) ainsi qu'une expression et qui renvoie le résultat de l'évaluation de l'expression. I.e.

```
eval :: Env -> Exp -> Val
```

et

```
eval pervasive sampleExp
```

renvoie `Vnum 7`.

3.5 Renommage α

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

```
λx -> λy ->
let f = λx -> x + 2 in
let g x = λg -> f (g x) in
let g (x, f) = f x
in λf -> g (x, f)
```

Renommer toutes les variables (e.g. en y ajoutant un 0, 1, 2, ...) pour que chaque variable ait un nom différent des autres. Bien sûr ce renommage ne doit pas changer la sémantique du code.