Programación Funcional

1er Parcial - MultiSets

Segundo semestre 2021

Los multisets son una estructura de datos que permiten saber cuántas veces fue ingresado un elemento. Para expresar sus operaciones, utilizaremos el lenguaje expresado por el tipo **`MSExp`**.

```
data MSExp a = EmptyMS
             | AddMS a (MSExp a)            | RemoveMS a (MSExp a)
             | UnionMS (MSExp a) (MSExp a)  | MapMS (a -> a) (MSExp a)
```

Las operaciones pretenden expresar las correspondientes operaciones de multisets.

- **`EmptyMS`** representa el multisets sin elementos (todos los elementos sin ocurrencias).
- **`AddMS`** representa el agregado de una ocurrencia de un elemento dado.
- **`RemoveMS`** representa la eliminación de una ocurrencia de un elemento dado.
- **`UnionMS`** representa la suma de ocurrencias de cada multiset para cada uno de sus elementos.
- **`MapMS`** representa la transformación de los elementos del multiset dado.

1. Definir por recursión explícita las siguientes funciones

a. **`occursMSE :: a -> MSExp a -> Int`**, que describe la cantidad de ocurrencias del elemento dado en el multiset.
AYUDA 1: como la función pedida NO se puede hacer por recursión estructural, considerar definir como función auxiliar **`occursMESWith :: a -> (a->a) -> MSExp a -> Int`** por recursión estructural e invocarla con los argumentos correspondientes
AYUDA 2: el número retornado NO puede ser negativo. O sea, los **`RemoveMS`** que no encuentran un **`AddMS`** correspondiente no deben tener efecto.

b. **`filterMSE :: (a -> Bool) -> MSExp a -> MSExp a`**, que describe el multiset resultante de eliminar todas las ocurrencias de los elementos que no cumplen con el predicado dado.
AYUDA: tener en cuenta que en el caso de **`MapMS`**, el elemento que se debe analizar es el procesado por la función dada en ese **`MapMS`**, y no el elemento sin procesar.

c. **`isValidMSE :: MSExp a -> Bool`**, que indica si la expresión de multiset dada es válida. Una expresión de multiset es inválida si tiene más **`RemoveMS`** que **`AddMS`** para un elemento determinado; en el caso de una **`UnionMS`**, cada parte se considera por separado.

d. **`evalMSE :: Eq a => MSExp a -> [a]`**, que describe la lista de todas las ocurrencias de los elementos del multiset dado.

e. **`simpMSE :: MSExp a -> MSExp a`**, que, suponiendo que recibe una expresión de multiset válida y describe al multiset resultante de simplificar el dado, aplicando las siguientes reglas en todos los lugares posibles:
   - **`(UnionMS EmptyMS e)`** se simplifica a **`e`**
   - **`(UnionMS e EmptyMS)`** se simplifica a **`e`**
   - **`(MapMS f EmptyMS)`** se simplifica a **`EmptyMS`**
   - **`(RemoveMS x (AddMS x e))`** se simplifica a **`e`**

2. Demostrar la siguiente propiedad: **`evalMSE . simpMSE = evalMSE`**.

3. Dar los tipos y escribir los esquemas de recursión estructural y primitiva para **`MSExp`**.

4. Escribir versiones de las funciones del ejercicio 1 sin utilizar recursión explícita.

Solución posible

1.
```
occursMSE x ms = occursMSE x id ms
occursMSEWith _ g EmptyMS         = 0
occursMSEWith x g (AddMS y e)     = delta (x==g y) + occursMESWith x g e
occursMSEWith x g (RemoveMS y e)  = let n = occursMSEWith x g e
                                    in n - delta ((n>0) && (x==y))
occursMSEWith x g (UnionMS e1 e2) = occursMSEWith x g e1
                                    + occursMSEWith x g e2
occursMSEWith x g (MapMS f e)     = occursMSEWith x (g . f) e

delta True  = 1
delta False = 0
```

2.
```
filterMSE _ EmptyMS         = EmptyMS
filterMSE p (AddMS x e)     = if p x then AddMS x (filter p e)
                                     else filter p e
filterMSE p (RemoveMS x e)  = if p x then RemoveMS x (filter p e)
                                     else filter p e
filterMSE p (UnionMS e1 e2) = UnionMS (filterMSE p e1)
                                      (filterMSE p e2)
filterMSE p (MapMS f e)     = MapMS f (filter (p . f) e)
```

3.
```
isValidMSE EmptyMS         = True
isValidMSE (AddMS _ e)     = isValidMSE e
isValidMSE (RemoveMS x e)  = ocurrsMSE x e > 0 && isValidMSE x
isValidMSE (UnionMS e1 e2) = isValidMSE e1 && isValidMSE e2
isValidMSE (MapMS _ e)     = isValidMSE e
```

4.
```
evalMSE EmptyMS         = []
evalMSE (AddMS x e)     = x : evalMSE e
evalMSE (RemoveMS x e)  = quitar x (evalMSE e)
evalMSE (UnionMS e1 e2) = evalMSE e1 ++ evalMSE e2
evalMSE (MapMS f e)     = map f (evalMSE e)

quitarSiHay x []     = error "No se puede quitar un elemento"
quitarSiHay x (y:ys) = if x==y then ys else y : quitar x ys
```

5.
```
simpMSE EmptyMSE         = EmptyMSE
simpMSE (AddMS x e)      = AddMS x (simpMSE e)
simpMSE (RemoveMS x e)   = simpRemoveMS x (simpMSE e)
simpMSE (UnionMS e1 e2)  = simpUnionMS (simpMSE e1) (simpMSE e2)
simpMSE (MapMS f e)      = simpMapMS f (simpMSE e)


simpRemoveMS x (AddMS y e) = if x==y
                             then e
                             else RemoveMS x (AddMS y e)
```

```
simpRemoveMS x e            = RemoveMS x e

simpUnionMS EmptyMS e2 = e2
simpUnionMS e1 EmptyMS = e1
simpUnionMS e1       e2 = UnionMS e1 e2

simpMapMS f EmptyMS = EmptyMS
simpMapMS f e       = MapMS f e
```

6. **PROP:** ¿ `evalMSE . simpMSE = evalMSE` ?

DEM: Por principio de extensionalidad, veremos que

¿para todo `e :: MSExp. (evalMSE . simpMSE) e = evalMSE e` ?

Por la definición de `(.)`, es equivalente ver que

¿para todo `e :: MSExp. evalMSE (simpMSE e) = evalMSE e` ?

Sea `e` una `MSExp`. Por principio de inducción en la estructura de `e`.

Caso base: `e=EmptyMS`)

¿ `evalMSE (simpMSE e) = evalMSE e` ?

Caso ind. 1: `e=AddMS x e1`)

HI1) ¡ `evalMSE (simpMSE e1) = evalMSE e1` !

TI) ¿ `evalMSE (simpMSE (AddMS x e1)) = evalMSE (AddMS x e1)` ?

Caso ind. 2: `e=RemoveMS x e1`)

HI1) ¡ `evalMSE (simpMSE e1) = evalMSE e1` !

TI) ¿ `evalMSE (simpMSE (RemoveMS x e1)) = evalMSE (RemoveMS x e1)` ?

Caso ind. 3: `e=UnionMS e1 e2`)

HI1) ¡ `evalMSE (simpMSE e1) = evalMSE e1` !

HI2) ¡ `evalMSE (simpMSE e2) = evalMSE e2` !

TI) ¿ `evalMSE (simpMSE (UnionMS e1 e2)) = evalMSE (UnionMS e1 e2)` ?

Caso ind. 4: `e=MapMS f e1`)

HI1) ¡ `evalMSE (simpMSE e1) = evalMSE e1` !

TI) ¿ `evalMSE (simpMSE (MapMS f e1)) = evalMSE (MapMS f e1)` ?

Caso base)

```
    evalMSE (simpMSE EmptyMS)
=               (por simpMSE.1)
    evalMSE EmptyMS
```

Caso ind. 1)

```
    evalMSE (simpMSE (AddMS x e1))
=               (por simpMSE.2)
    evalMSE (AddMS x (simpMSE e1))
=               (por evalMSE.2)
    x : evalMSE (simpMSE e1)
=               (por HI)
    x : evalMSE e1
=               (por evalMSE.2)
    evalMSE (AddMS x e1)
```

Caso ind. 2)

```
    evalMSE (simpMSE (RemoveMS x e1))
```

```
=                (por simpMSE.3)
    evalMSE (simpRemoveMS x (simpMSE e1))
=                (por Lema evalMSE-simpRemoveMS)
    quitar x (evalMSE (simpMSE e1))
=                (por HI)
    quitar x (evalMSE e1)
=                (por evalMSE.3)
    evalMSE (RemoveMS x e1)
```

Caso ind. 3)

```
    evalMSE (simpMSE (UnionMS e1 e2))
=                (por simpMSE.4)
    evalMSE (simpUnionMS (simpMSE e1) (simpMSE e2))
=                (por Lema evalMSE-simpUnionMS)
    evalMSE (simpMSE e1) ++ evalMSE (simpMSE e2)
=                (por HI1 y HI2)
    evalMSE e1 ++ evalMSE e2
=                (por evalMSE.4)
    evalMSE (UnionMS e1 e2)
```

Caso ind. 4)

```
    evalMSE (simpMSE (MapMS f e1))
=                (por simpMSE.5)
    evalMSE (simpMapMS f (simpMSE e1))
=                (por Lema evalMSE-simpMapMS)
    map f (evalMSE (simpMSE e1))
=                (por HI)
    map f (evalMSE e1)
=                (por evalMSE.5)
    evalMSE (MapMS f e1)
```

LEMA `evalMSE-simpRemoveMS`)

¿ `evalMSE (simpRemoveMS x e) = quitar x (evalMSE e)` ?

LEMA `evalMSE-simpUnionMS`)

¿ `evalMSE (simpUnionMS e1 e2) = evalMSE e1 ++ evalMSE e2` ?

LEMA `evalMSE-simpMapMS`)

¿ `evalMSE (simpMapMS f e) = map f (evalMSE e)` ?

Dem: sean `f` y `e`, cualesquiera.

Caso 1: `e = EmptyMS`)

¿ `evalMSE (simpMapMS f EmptyMS) = map f (evalMSE EmtpyMS)` ?

Caso 2: `e` distinto de `EmptyMS`)

¿ `evalMSE (simpMapMS f e) = map f (evalMSE e)` ?

7. `fold`
8. `is`