Pràctica de Haskell.
Programes funcionals simples: execució i comprovació de tipus

1 Presentació

Per a resoldre els problemes de les seccions 2 i 3 només es poden usar funcions de l'entorn `Prelude`.

Volem crear el nostre propi llenguatge funcional, que s'executa de forma similar a Haskell. També volem ser capaços d'inferir el tipus de les nostres expressions i funcions. Treballarem amb dos tipus de termes: els que usarem per executar (`Term`) i els que usarem per fer la comprovació dels tipus (`TTerm`). També tindrem un `data` per representar els tipus de les expressions dels nostres programes. El programes els expressarem amb equacions que defineixen funcions i agruparem les equacions que defineixen la mateixa funció. D'aquesta manera un programa es una llista de llistes de parells d'expressions. Si ho definim genèric en el tipus d'expressions tenim:

```
data Program a = Prog [[(a,a)]]
```

Afegiu que sigui de les classes Show, Read i Eq usant el `deriving`

Per a la pràctica considerarem que tenim un `Program Term` donat i l'usarem per avaluar altres `Term` i per fer comprovació de tipus. En la primera part del document us demanarem que resolgueu diferent problemes sobre el tipus `Term`. En la segona part transformarem els `Term` en `TTerm` on les operacions estan currificades i farem comprovació de tipus. A la tercera part caldrà fer un executable i afegir una funció que executi els programa utilitzant aleatorització.

2 Execució de programes

Com hem indicat en aquest apartat treballarem amb objectes del tipus `Program Term`, és a dir que les equacions estan formades per objectes del data Term (que no és polimòrfic). Per a definir aquest data considereu els següents constructors:

- `Num` que té un enter com argument
- `Var` que rep un string i representa una variable amb el string com a nom.

- `ITE` que té tres `Term` com arguments i representa un if-then-else.
- `LET` que rep un string i dos `Term` i representa el let on les variables amb el string donat que apareixen al segon `Term` han de ser reemplaçades pel primer `Term`.
- `Func` que rep un string i una llista de `Term` i representa la funció amb el nom del string i amb els arguments de la llista.

Aquí teniu alguns exemples de valors del tipus `Term` i un programa de tipus `(Program Term)` que defineix la concatenació de llistes:

```
e11 = Func "Append" [Func "Empty" [],Var "l"]
e12 = Var "l"
e21 = Func "Append" [Func "Cons" [Var "x", Var "l1"], Var "l2"]
e22 = LET "m" (Func "Append" [Var "l1", Var "l2"]) (Func "Cons" [Var "x", Var "m"])
prog1 = Prog [[(e11,e12),(e21,e22)]]
```

A cada parell de termes li direm *equació*. Així el program està format per equacions que agrupem per la funció que defineixen. Els programes que heu de tractar satisfan les següents condicions:

- Per a cada llista d'equacions dins del programa, el terme de l'esquerra de cada equació està encapçalat per `Func` amb el mateix string (i la mateixa quantitat d'arguments). Cada llista defineix una funció diferent.
- Als termes de l'esquera de les equacions cada (nom) de variable apareixerà un sol cop en el terme. A la dreta no hi ha restriccions. Això és igual que a les equacions de Haskell.
- Als termes de l'esquerra de les equacions només apareixeran els constructors `Var` i `Func`.
- Totes les variables lliures (que no apareixen en un `LET`) que trobem al terme de la dreta, també apareixen al terme de l'esquerra.
- Considerem que tenim les següents funcions predefinides i que no apareixeran mai a nivell superior d'un terme de l'esquerra d'una equació: `">"`, `"=="`, `"not"`, `"and"`, `"or"`, `"True"`, `"False"`, `"+"`, `"−"`, `"∗"` i `"Empty"` i `"Cons"` per llistes d'enters.

Feu els següents apartats.

1. Definiu el data Term i feu que sigui de la classe Show i Read amb deriving.
2. Una *substitució* és una llista de parells `(String,Term)` que ens diu que hem de substituir cada variable amb un string a la llista pel terme corresponent. Feu una funció `replace::[(Sting,Term)] -> Term -> Term`, que donada una substitució i un terme retorna el terme resultant d'aplicar la substitució (és a dir, reemplaça totes les aparicions de `Var` amb un strings dins de la substitució pel terme corresponent).

3. Feu que Term sigui `instance` de la `class Eq` on dos termes són iguals si son idèntics renombrant el nom de les variables dels `LET`. Per exemple si tenim:

```
ne1 = LET "x" (Num 3) (LET "y" (Num 5) (Func "+" [Var "x", Var "y"]))
ne2 = LET "y" (Num 3) (LET "x" (Num 5) (Func "+" [Var "x", Var "y"]))
ne3 = LET "y" (Num 3) (LET "x" (Num 5) (Func "+" [Var "y", Var "x"]))
```

   Llavors `ne1` i `ne2` són diferents i `ne1` i `ne3` són iguals.

4. Definiu una operació `match::Term -> Term -> Maybe [(String,Term)]` que donats dos termes ens diu si fan *match* com a Haskell. El primer terme sempre serà la part esquerra d'una equació de programa (mireu les condicions que satisfà). Si no fan match retorna `Nothing`. En altre cas torna `Just` de la substitució que aplicada al primer terme el fa igual al segon. Com a exemple si

```
exm1 = Func "Append" [Func "Cons" [Var "x", Var "l1"], Var "l2"]
exm2 = Func "Append" [Func "Cons" [Num 3, Func "Empty" []], Func "Empty" []]
```

   llavors `match exm1 exm2` és

```
Just [("x",Num 3),("l1",Func "Empty" []),("l2",Func "Empty" [])]
```

5. Definiu la funció `oneStep::Program Term -> Term -> Term` que rep un programa i un terme i retorna el terme resultant d'aplicar un pas d'avaluació. El pas s'aplica en la primera posició que trobem en postordre (primer mirem els arguments d'esquerra a dreta i després l'arrel). Noteu que aquest no és l'ordre d'avaluació que usa Haskell. Aplicar un pas significa:

   - si tenim el constructor `Func` i l'operació és predefinida tenim les següents opcions: (i) si l'operació és `">"`, `"=="`, `"+"`, `"−"`, `"∗"` la podem resoldre si els dos arguments són `Num` operant amb els enters; (ii) si és `"not"` la podem resoldre si l'argument és `Func "True"[]` o `Func "False"[]`; (iii) si és `"and"` , `"or"` la podem resoldre si un dels dos arguments és un Term que representa True o False (com el cas anterior).
   - si tenim un `LET` substituirem la variable pel primer terme dins del segon terme.
   - si tenim un `ITE` el podem reduir si el primer terme representa True o False (com al primer cas) i el reemplacem pel segon o el tercer terme segons el que sigui el primer.
   - Si és un `Func` amb una operació no predefinida buscarem una equació del programa que la part esquerra faci match i el reemplaçarem per la part dreta aplicant la substitució que ens dona el match.

Si no és pot aplicar cap pas, el resultat és el mateix terme que ens han passat. Per exemple,

```
oneStep prog1 (Func "Append" [Func "Cons" [ne1, Func "Empty" []], Func "Empty" []])
```

retorna

```
Func "Append" [Func "Cons" [LET "x" (Num 3) (Func "+" [Var "x", (Num 5)]), Func "Empty" []], Func "Empty" []]
```

i si seguim aplicant `oneStep prog1` al resultat obtindríem consecutivament

```
Func "Append" [Func "Cons" [Func "+" [(Num 3), (Num 5)], Func "Empty" []], Func "Empty" []]
Func "Append" [Func "Cons" [Num 8, Func "Empty" []], Func "Empty" []]
LET "m" (Func "Append" [Func "Empty" [], Func "Empty" []]) (Func "Cons" [Num 8, Var "m"]
LET "m" (Func "Empty" []) (Func "Cons" [Num 8, Var "m"]
Func "Cons" [Num 8, Func "Empty" []]
```

6. Definiu la funció `reduce:: Program Term -> Term -> Term`, que aplica `oneStep` fins que ja no canvia el terme. En l'exemple anterior aplicat al terme inicial obtindríem

   ```
   Func "Cons" [Num 8, Func "Empty" []]
   ```

3 Comprovació de tipus

Considereu ara el nou data `TTerm`, que té els següents constructors:

- `INum` que té un enter com argument
- `IFunc` que rep un string i representa una funció (sense aplicar-la a cap argument).
- `IVar` que rep un string i representa una variable amb el string com a nom.
- `Apply` que rep dos `TTerm` i representa l'aplicació com a Haskell.
- `Lambda` que rep un string i un `TTerm` i representa un lambda terme com a Haskell.

Per aquesta part a més de les funcions predefinides considerades en l'anterior apartat també usarem la funció `"ITE"` per representar el if-then-else.

7. Definiu el data `TTerm` i feu que sigui de la classe Show i Read amb deriving.
8. Feu la funció `transform:: Term -> TTerm` que transforma un `Term` en un `TTerm` de manera que totes les operacions es tractin currificadament (usant l'aplicació) i el LET es transforma usant la aplicació i la lambda. Com exemple tenim que transform aplicat a `e22` ens dona

```
Apply (Lambda "m" (Apply (Apply (IFunc "Cons") (IVar "x")) (IVar "m")))
(Apply (Apply (IFunc "Append") (IVar "l1")) (IVar "l2"))
```

Feu la funció `transformProgram:: Program Term -> Program TTerm` que aplica la transformació anterior a tots el termes del programa d'entrada.

Considerem el data `Type` per a representar els tipus dels `TTerm` amb els següents constructors:

- TBool per expressar el tipus Bool
- TInt per expressar el tipus Int
- ListInt per expressar el tipus llista de Int
- TVar que rep un string i representa una variable de tipus amb el string com a nom.
- Arrow de dos `Type` per expressar el tipus funció (la `->` de Haskell).

Considerant els següents tipus per a les funcions predefinides:

- `">", "==":(Arrow TInt (Arrow TInt TBool))`
- `"not":(Arrow TBool TBool)`
- `"and", "or":(Arrow TBool (Arrow TBool TBool))`
- `"True", "False":TBool`
- `"+", "−", "∗":(Arrow TInt (Arrow TInt TInt))`
- `"Empty":ListInt, "Cons":(Arrow TInt (Arrow ListInt ListInt))`
- `"ITE":(Arrow TBool (Arrow (TVar "a") (Arrow (TVar "a") (TVar "a"))))`

Per `Lambda` i `Apply` heu d'usar els mateixos tipus que per la lambda i la aplicació de Haskell.

9. Feu la funció `wellTyped:: Program TTerm -> Bool` que donat programa ens diu si totes les equacions del programa tenen un tipus correcte y el mateix per a totes les equacions de cada definició. Seguiu l'algorisme (simplificat) de Hindley-Milner vist a classe, per fer la comprovació.

4 Entrada/Sortida y aleatorització.

Feu un main Haskell (i les funcions auxiliars que calguin) que llegeix un programa ja amb el format per a poder llegir-lo amb un `read` que està en la primera línia de l'arxiu "programhs.txt" (sempre tindrà aquest nom) i que a continuació ens diu si el programa tipa o no. A continuació ens demana per l'entrada estàndard un string que representa un `Term` que estarà ben escrit per a que el `read` el pugui convertir en `Term`. Llavors en demanarà si volem a si el reduir el terme amb la estratègia postorde, és a dir usar el `reduce`, o be amb una estratègia aleatòria. Que vol dir que cada pas s'aplica en una posició escollida al atzar (usant el Random). Per tant, cal definir una última funció `reduceRandom` que ens permeti reduir un `Term` escollint aleatòriament la posició on es redueix a cada pas.

Excepte el programa a interpretar tota la resta de dades es llegiran per l'entrada estàndard.
Per a poder llegir d'un fitxer caldrà importar el mòdul `System.IO`, posant

```
import System.IO
```

i usar les operacions `openFile` (amb `ReadMode`), `hGetLine` i `hClose`, vistes a classe.