Pràctica obligatòria de Haskell. Reescriptura

1 Presentació

Volem implementar un sistema per treballar de forma general amb *regles de reescriptura*. La reescriptura és un mecanisme semblant al de l'execució de programes Haskell. En lloc d'equacions com a Haskell (que defineixen el programa) tenim regles que indiquen què es canvia per què. Per exemple,

$$aa \longrightarrow aba$$

és una regla sobre *strings* que indica que si trobo dins d'un string el substring aa el puc reemplaçar per el substring aba (sigui on sigui).

El fet de que s'anomenin regles o equacions no és important ja que sempre s'apliquen substituint l'esquerra per la dreta (tan en reescriptura com en Haskell).

La diferència principal és que la reescriptura es pot aplicar a objectes generals, és abstracte, i pot seguir qualsevol *estratègia* d'aplicació de les regles, que decideix l'usuari (no com a Haskell on l'estratègia per aplicar les equacions és fixa i està predefinida).

Els objectes a reescriure estan definits sobre una *Signature* o vocabulari, que ens diu quins símbols tenim i quants arguments poden rebre (que pot ser zero).

Els objectes de les regles poden tenir, a més, variables que s'usen per indicar que allà hi pot anar qualsevol cosa. Per exemple, la regla

$$f(x, x) \longrightarrow g(g(x))$$

ens permet reescriure el terme $h(f(a, a))$ en $h(g(g(a)))$.

En aquest exemple x és una variable. Noteu que x surt més d'un cop a la par esquerra, cosa que no està permesa a Haskell. També noteu que **no** usem les funcions currificades, és a dir, les funcions sempre reben tots els paràmetres. **Tampoc tindrem funcions d'ordre superior**.

L'objectiu de la pràctica és definir la reescriptura usant una *class* de Haskell, i després particularitzar (`instance`) aquesta classe de dues maneres diferents per obtenir reescriptura de *strings* i reescriptura d'*arbres* (o termes), amb diferents estratègies.

Per aconseguir-ho heu de seguir les següents passes. **És obligatori usar els noms per les classes, tipus i les funcions que s'indica i els tipus que s'indica que han de tenir**.

2 La `class Rewrite`

Com s'ha dit, una `Signature` és una llista de parells de `String` i `Int`, que indica quins són els identificadors per a construir els objectes a reescriure i quants arguments reben.

Addicionalment, definim també una `Position` en un objecte com una lista de `Int` que ens indiquen una posició dins de l'objecte. Per exemple, en un String ens indica l'inici d'un sufix i en un arbre ens indica un subarbre determinat. En aquest darrer cas, si considerem que el primer fill és el fill 0, la llista [0,2,0] indica que hem d'anar al primer fill, després al tercer i un altre cop al primer per trobar el subarbre. La llista buida indica la posició arrel.

Finalment, ens cal definir que és una `Substitution` d'objectes de tipus `a`, que representem amb una llista de parells de tipus `a`, on a cada parell el primer element és una variable i el segon és el terme pel que s'ha de substituir la variable.

Amb tot això, tenim que el tipus `a` és de la `class Rewrite` si té les següents operacions:

1. `getVars`, que rep un objecte de tipus `a` i retorna una llista d'objectes de tipus `a`, que són les variables que apareixen a l'objecte rebut.

2. `valid`, que rep una `Signature` i un objecte de tipus `a` i retorna un `Bool`, que indica si l'objecte està construït correctament seguint la signatura.

3. `match`, que rep dos objectes de tipus `a` i retorna una llista de parell `(Position,Substitution a)` que indica en quines posicions del segon objecte el primer fa *matching* amb la corresponent substitució. És a dir, que aplicant la substitució al primer tenim exactament els subobjecte que està a la posició.

4. `apply`, que rep un objecte de tipus `a` i una substitució d'objectes de tipus `a` i retorna l'objecte resultant de substituir les variables pels objectes de la substitució.

5. `replace`, que rep un objecte de tipus `a` i una llista de parells de Position i objecte de tipus `a` i retorna el resultat de canviar cada subobjecte del primer paràmetre en una de les posicions donades per l'objecte que l'acompanya. En aquesta funció podeu assumir que cap de les posicions donades es solapa.

   Noteu que dues posicions són solapades quan una indica un subobjecte de l'altre.

6. `evaluate`, que rep un objecte de tipus `a` i retorna un objecte de tipus `a`, que és el resultat d'haver avaluat l'objecte segons algunes símbols predefinits, que no es tracten amb regles de reescriptura. Per exemple si tenim sumes i productes i números dins els nostres objectes la funció evaluate resoldria les sumes o productes que s'apliquen a números. Si no hi ha símbols predefinits, simplement és la identitat.

3 Regles, estratègies i reescriptura

Una *regla de reescriptura* està formada per dos objectes i un *sistema de reescriptura* és un conjunt de regles. Definiu un `data Rule` (que depén del tipus dels objectes que reescriu) per les regles i un `data` o `type RewriteSystem` per als sistemes de regles.

Feu que `Rule` sigui `instance` de la classe `Show` escrivint les regles posant l'objecte de l'esquerra separat del de la dreta per `" --> "` (sense que surtin les cometes). Si heu fet que `RewriteSystem` sigui un `data` feu el mateix.

1. Una regla és vàlida respecte a una signatura si els dos objectes són vàlids respecte a la signatura i les variables de l'objecte de la dreta estan incloses en les de l'objecte de l'esquerra. Un sistema es vàlid si totes les regles són valides.

   Feu les funcions `validRule` i `validRewriteSystem` que reben una signatura i una regla o un sistema de regles segon el cas (en aquest ordre) i retornen un `Bool` que indica si són vàlids o no. Les regles han de ser d'objectes de la classe `Rewrite`.

2. Una estratègia és una funció que rep una llista de parells de posició i objecte (com el segon que rep el `replace`) i retorna una llista del mateix tipus on només alguns dels parells s'han guardat.

   Com que la llista serà usada pel `replace`, hem de garantir que la llista resultant no conté posicions solapades.

3. Hem d'implementar la reescriptura genèrica d'un objecte amb un sistema de regles seguint una estratègia determinada. Per això usarem la `class Rewrite`.

   (a) Definiu la funció `oneStepRewrite` que rep un sistema, un objecte i una estratègia (en aquest ordre) i retorna l'objecte resultant de fer un pas de reescriptura. Com que ho volem fer general, per fer això cal primer obtenir totes les posicions on alguna regla es pot aplicar i usar l'estratègia per filtrar-les. És a dir, un pas pot reemplaçar més d'un subobjecte, però sempre en posicions no solapades.

   (b) Usant la funció anterior i la funció `evaluate` definiu la funció `rewrite` que rep un sistema, un objecte i una estratègia (en aquest ordre) i retorna l'objecte de resultant d'aplicar repetidament un pas de reescriptura seguit de l'`evaluate` fins que no es pugui més. Noteu que no està garantit que acabi.

   Definiu també la funció `nrewrite` que rep, a més, un `Int` com a quart paràmetre (que serà no negatiu), i que aplica tantes passes de reescriptura (més evaluate) com indica el número que rebem.

4 Reescriptura de strings

Anem a definir la reescriptura de strings usant la classe `Rewrite`. Per això feu les següents coses.

1. Definiu un data `RString` que representi els strings per reescriptura i feu que sigui `instance` de la classe `Rewrite`. Noteu que com que el `replace` demana que les posicions no estiguin solapades, en el cas de reescriptura de strings només podem tenir una posició en la llista.

   Considereu que els símbols dels strings són una lletra entre `'a'` i `'z'` seguida per 0 o més digits (entre `'0'` i `'9'`). És a dir, que el string `"a1b2a13"` en realitat té tres elements el `"a1"`, el `"b2"` i el `"a13"`.

2. Feu l'operació `readRString :: String -> RString`, que llegeix un string normal de Haskell i el converteix en un `RString`. També feu l'operació readRStringSystem que rep una llista de parells de `String` i retorna un `RewriteSystem` de `RString`.

3. Feu que `RString` sigui `instance` de la classe `Show` amb el show que no mostri el constructor, és a dir només el string (però sense cometes).

4. Definiu dues estratègies:

   - *leftmost*, que reescriu la posició més a l'esquerra possible.
   - *rightmost*, que reescriu la posició més a la dreta possible.

5 Reescriptura d'arbres

Considerem ara la reescriptura d'arbres (o termes) amb els enters positius amb suma i producte com a predefinits.

1. Definiu un data `RTerm` que representi els arbres per reescriptura. Podeu usar constructors per distingir els símbols normals de les variables, ja que els dos els representarem amb `Strings` i dels números. Per a la suma i el producte feu el que preferiu. Feu que `RTerm` sigui `instance` de la classe `Rewrite`.

2. Feu que `RTerm` sigui `instance` de la classe `Show` amb el show que mostri el terme sense els constructors i sense les cometes als strings i amb notació no currificada, és a dir, si un símbol té un o més arguments han de sortir entre parèntesi i separats per comes (com als exemples de la Secció 1).

3. Feu l'operació `readRTree :: [(String,Int)] -> RTerm`, que converteix la llista d'entrada en un `RTerm` seguint la següent convenció:

   - si trobeu el parell `(s,-1)` vol dir que `s` és una variable.
   - si trobeu el parell `(s,-2)` vol dir que `s` és una número (és a dir un string només amb dígits).

- si trobeu el parell `(s,n)` amb $n \geq 0$ vol dir que `s` és un símbol i n és el nombre d'arguments que té. En aquest cas, després d'aquest parell venen, del primer al darrer, els `n` fills (arbres) d'aquest símbol (que podeu obtenir llegint recursivament).

Podeu assumir que la llista que obteniu representa un RTerm (encara que podria no ser vàlid respecte a una `Signature` donada). Per exemple, el show del readRTree de la llista:

```
[("hol",3),("+",2),("32",-2),("g8",1),("x",-1),("a2",0),("fis",2),("x",-1),("y",-1)]
```

és

```
"hol( +( 32 , g8( x ) ) , a2 , fis( x , y ) )"
```

També feu l'operació readRTermSystem que rep una llista de parells de `[(String,Int)]`, és a dir `[([(String,Int)],[(String,Int)])]`, i retorna un `RewriteSystem` de `RTerm`.

4. Definiu quatres estratègies:

   - *parallelinnermost*, que reescriu totes les posició més internes. És a dir, totes les que no tenen cap posició més interna que també es pot reescriure.
   - *leftmostinnermost*, que reescriu la posició més interna i més a l'esquerra possible. És a dir, d'entre totes les que no tenen cap posició més interna que també es pot reescriure, ens quedem la que sigui més a l'esquerra.
   - *paralleloutermost*, que reescriu totes les posició més externes. És a dir, totes les que no tenen cap posició superior que també es pot reescriure.
   - *leftmostoutermost*, que reescriu la posició més externa i més a l'esquerra. És a dir, d'entre totes les que no tenen cap posició més externa que també es pot reescriure, ens quedem la que sigui més a l'esquerra.