Llenguatges de Programació

Raonament equacional

Jordi Petit

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

Contingut

Raonament equacional

El **raonament equacional** permet reflexionar sobre programes funcionals per tal d'establir propietats usant igualtats i substitucions matemàtiques.

Sovint s'estableixen equivalències entre funcions.

Aquestes es poden aprofitar per:

- millorar l'eficiència de programes.

- verificar programes:
  demostrar que un programa és correcte respecte la seva especificació.

- derivar programes:
  deduir el programa formalment a partir de l'especificació.

Exemple: Multiplicació de complexos

$(a + bi) * (c + di)$ es pot calcular amb 4 productes de reals:

```
def mult(a, b, c, d):
    re1 = a*c - b*d
    im1 = b*c + a*d
    return (re1, im1)
```

Gauss va descobrir que només calien 3 productes:

```
def gauss(a, b, c, d):
    t1 = a * c
    t2 = b * d
    re2 = t1 - t2
    im2 = (a + b) * (c + d) - t1 - t2
    return (re2, im2)
```

Podem comprovar matemàticament l'equivalència entre les dues funcions:

```
(re2, im2) = (t1 - t2, (a + b) * (c + d) - t1 - t2)
           = (a*c - b*d, (a + b) * (c + d) - a*c + b*d)
           = (re1, a*c + a*d + b*c + b*d - a*c - b*d)
           = (re1, b*c + a*d)
           = (re1, im1)
```

Contingut

Associativitat de la composició

Propietat: `f . (g . h) = (f . g) . h`.

Definició:

```
(.) :: (b -> c) -> (a -> b) -> (a -> c)
(f1 . f2) x = f1 (f2 x)                                    ⭐
```

Demostració: Sigui qualsevol dada `x`, llavors:

```
(f . (g . h)) x =
        -- definició de .
    = f ((g . h) x)
        -- definició de .
    = f (g (h x))
        -- definició de .
    = (f . g) (h x)
        -- definició de .
    = ((f . g) . h) x
```

```
⇒   f . (g . h) = (f . g) . h
```

Observació: S'ha usat ⭐ en els dos sentits.

Involució de la negació

Propietat: `not . not = id`.

```haskell
not :: Bool -> Bool
not True = False                                1️⃣
not False = True                                2️⃣

id :: a -> a
id x = x                                        3️⃣
```

Demostració: Hi ha dos casos:

```haskell
(not . not) True =
        -- definició de .
    = not (not True)
        -- 1️⃣
    = not False
        -- 2️⃣
    = True
        -- 3️⃣
    = id True
```

Involució de la negació

```
(not . not) False =
        -- definició de .
    = not (not False)
        -- 2️⃣
    = not True
        -- 1️⃣
    = False
        -- 3️⃣
    = id False
```

```
⇒   not . not = id
```

Observació: S'han cobert tots els possibles casos explícitament.

Contingut

`map` és distributiva sobre `++`

Propietat: `map f (xs ++ ys) = map f xs ++ map f ys`.

```
map f [] = []                       1️⃣
map f (x:xs) = f x : map f xs       2️⃣

[] ++ ys = ys                       3️⃣
(x:xs) ++ ys = x : xs ++ ys         4️⃣
```

Demostració: Inducció sobre `xs`.

🅰️ Cas base: `xs = []`.

```
map f ([] ++ ys) =
        -- 3️⃣
    = map f ys
        -- 3️⃣
    = [] ++ map f ys
        -- 1️⃣
    = map f [] ++ map f ys
```

`map` és distributiva sobre `++`

Demostració: Inducció sobre `xs`.

🅱️ Cas inductiu: `xs = z:zs`. HI: `map f (zs ++ ys) = map f zs ++ map f ys`.

```haskell
map f (xs ++ ys) =
        -- definició de xs
    = map f ((z:zs) ++ ys)
        -- 4️⃣
    = map f (z : zs ++ ys)
        -- 2️⃣
    = f z : map f (zs ++ ys)
```

```haskell
map f xs ++ map f ys =
        -- definició de xs
    = map f (z:zs) ++ map f ys
        -- 2️⃣
    = (f z : map f zs) ++ map f ys
        -- 4️⃣
    = f z : (map f zs ++ map f ys)
        -- hipotesi d'inducció
    = f z : map f (zs ++ ys)
```

```haskell
⇒   map f (xs ++ ys) = map f xs ++ map f ys
```

`map` és distributiva sobre `++`

Demostració:

S'ha demostrat per inducció sobre `xs`:

🅰️ Cas base: `xs = []`.

🅱️ Cas inductiu: `xs = z:zs`.

Per tant,

```
map f (xs ++ ys) = map f xs ++ map f ys
```

Involució del revessat

Propietat: `reverse . reverse = id`.

```
reverse [] = []
reverse (x:xs) = reverse xs ++ [x]
```

Lema 1:

```
reverse [x] = [x]
```

Demostració: exercici (fàcil!).

Lema 2:

```
reverse (xs ++ ys) = reverse ys ++ reverse xs
```

Demostració: exercici (inducció sobre `xs`).

Involució del revessat

Propietat: `reverse . reverse = id`.

```haskell
reverse [] = []
reverse (x:xs) = reverse xs ++ [x]
```

Demostració:

🅰️ Cas base: `xs = []`.

```haskell
(reverse . reverse) [] =
        -- definició de .
    = reverse (reverse [])
        -- definició de reverse
    = reverse []
        -- definició de reverse
    = []
        -- definició de id
    = id []
```

Involució del revessat

🅱️ Cas inductiu: `xs = z:zs`.

Hipòtesi d'inducció: `(reverse . reverse) zs = id zs`

```
(reverse . reverse) xs =
        -- definició de xs
    = (reverse . reverse) (z:zs)
        -- definició de .
    = reverse (reverse (z:zs))
        -- definició de reverse
    = reverse (reverse zs ++ [z])
        -- lema 2
    = reverse [z] ++ reverse (reverse zs)
        -- lema 1
    = [z] ++ reverse (reverse zs)
        -- definició de .
    = [z] ++ (reverse . reverse) zs
        -- hipotesi d'inducció i definició de id
    = [z] ++ zs
        -- definició de ++
    = z:zs
        -- definició de xs
    = xs
```

Involució del revessat

Per tant, queda demostrat que:

```
reverse . reverse = id
```

tal com es volia.

Contingut

map per arbres

```haskell
data Arbin a = Empty | Node a (Arbin a) (Arbin a)

treeMap :: (a -> b) -> (Arbin a) -> (Arbin b)

treeMap _ Empty = Empty
treeMap f (Node x l r) = Node (f x) (treeMap f l) (treeMap f r)
```

Propietat: `treeMap id = id`.

Demostració: Inducció sobre l'arbre `t`:

🅰️ Cas base: `t = Empty`.

```haskell
treeMap id Empty =
        -- definició de treeMap
    = Empty
        -- definició de id
    = id Empty
```

map per arbres

Demostració: Inducció sobre l'arbre `t`:

🅱️ Cas inductiu: `t = Node x l r`.

HI: `treeMap id l = l` i `treeMap id r = r`

```
treeMap id t =
        -- definició de t
    = treeMap id (Node x l r)
        -- definició de treeMap
    = Node (id x) (treeMap id l) (treeMap id r)
        -- definició de id
    = Node x (treeMap id l) (treeMap id r)
        -- HI
    = Node x l r
        -- definició de t
    = t
```

Per tant,

`treeMap id = id`.

Contingut

- Introducció
- Base
- Inducció
- Inducció amb arbres
- Millora eficiència
- Sumari
- Exercicis

Millorant `reverse`

Especificació:

```
reverse :: [a] -> [a]

reverse [] = []                          1️⃣
reverse (x:xs) = reverse xs ++ [x]       2️⃣
```

Problema: Donat que `++` necessita temps $O(n)$, `reverse` necessita temps $O(n^2)$ 🙁.

Podem revessar més ràpidament? 🤔

Millorant `reverse`

Considerem una *generalització* de `reverse`:

```
revcat :: [a] -> [a] -> [a]

revcat xs ys = reverse xs ++ ys                3️⃣
```

Llavors:

```
reverse xs = revcat xs []
        -- per 3️⃣ i per definició de ++
```

Podem ara definir `revcat`? 🤔

Millorant `reverse`

🅰️ Cas base: `xs = []`.

```
revcat [] ys =
        -- 3️⃣ i definició de ++
    = reverse [] ++ ys
        -- 1️⃣
    = [] ++ ys
        -- definició de ++
    = ys
```

🅱️ Cas inductiu: `xs = z:zs`.

```
revcat (z:zs) ys =
        -- 3️⃣
    = reverse (z:zs) ++ ys
        -- 2️⃣
    = (reverse zs ++ [z]) ++ ys
        -- associativitat de ++
    = reverse zs ++ ([z] ++ ys)
        -- 3️⃣
    = revcat zs ([z] ++ ys)
        -- lema sobre ++
    = revcat zs (z:ys)
```

Millorant `reverse`

Per tant,

```haskell
reverse xs = revcat xs []
    where
        revcat [] ys = ys
        revcat (x:xs) ys = revcat xs (x:ys)
```

funciona en temps lineal! 😀

A més, `revcat` és recursiva final, per tant necessita espai constant.

- Una funció recursiva és *final* si la crida recursiva és el darrer pas que fa ➡ es pot canviar el `call` per un `jmp`.

Contingut

Sumari

La programació funcional permet raonar senzillament sobre els programes usant equacions i mètodes matemàtics.

Trobar equivalències entre expressions:

- Pot ser útil per demostrar propietats i correctesa.
- Pot ser útil per donar lloc a optimitzacions.

Demostrar equivalències vol pràctica (igual que la programació).

Per fer-ho amb èxit sovint cal:

- ser exhaustiu.
- usar definicions en els dos sentits.
- utilitzar inducció.
- aprofitar les definicions recursives.
- demostrar resultats auxiliars.

Lectura recomanada:

- Graham Hutton. A tutorial on the universality and expressiveness of fold. Journal on Functional Programming, 1999.

Contingut

Exercicis

(Assumiu que totes les EDs són finites i els tipus correctes)

1. Demostreu que `xs ++ [] = xs = [] ++ xs`.
2. Demostreu que `xs ++ (ys ++ zs) = (xs ++ ys) ++ zs`.
3. Demostreu que `reverse (xs ++ ys) = reverse ys ++ reverse xs`.
4. Demostreu que `length (xs ++ ys) = length xs + length ys.`.
5. Demostreu que `take n xs ++ drop n xs = xs`.
6. Demostreu que `drop n (drop m xs) = drop (m + n) xs`.
7. Demostreu que `head . map f = f . head`. Aneu amb compte amb el cas de la llista buida. Quin interès té aquesta equivalència?
8. Demostreu que `filter p (xs ++ ys) = filter p xs ++ filter p ys`.
9. Demostreu que `foldl f e xs = foldr (flip f) e (reverse xs)`.
10. Demostreu que `foldl (@) e xs = foldr (<>) e xs` quan `(x <> y) @ z = x <> (y @ z)` i `e @ x = x <> e`.

Exercicis

(Assumiu que totes les EDs són finites i els tipus correctes)

1. Considereu aquest tipus pels naturals:

```
data Nat = Z | S Nat
```

Definiu les funcions següents amb significat obvi:

```
intToNat :: Int -> Nat
natToInt :: Nat -> Int
add      :: Nat -> Nat -> Nat     -- no es pot usar la suma d'Ints!.
```

- Demostreu que `intToNat` i `natToInt` són inverses l'una de l'altra.
- Demostreu que `Z` és element neutre (per la dreta i per l'esquerra) de `add`.
- Demostreu que `add (S x) y = add x (S y)`.
- Demostreu l'associativitat de `add`.
- Demostreu la commutativitat de `add`.

Exercicis

(Assumiu que totes les EDs són finites i els tipus correctes)

1. Definiu arbres binaris amb una operació `size` i una operació `mirror`. Demostreu que `size . mirror = size`.

Exercicis

Una llista es diu que és *supercreixent* si cada element és més gran que la suma dels seus anteriors:

```
superCreixent :: Num a, Ord a => [a] -> Bool

superCreixent [] = True
superCreixent (x:xs) = superCreixent && x > sum xs
```

1. Mostreu que `superCreixent` funciona en temps quadràtic.

2. Definiu una funció `superCreixent' :: [a] -> (Bool, a)` que, donada una llista, retorni si és supercreixent *i* quina és la seva suma.

3. Escriviu `superCreixent'` en termes de `superCreixent`.

4. Escriviu `superCreixent` en termes de `superCreixent'`.

5. Deriveu `superCreixent'`.

6. Doneu el temps d'execució de `superCreixent'`.