Programação Funcional – Exercícios de indução

Sandra Alves
DCC/FCUP

2017/18

Associatividade da adição
Caso base

```
  Zero + ( y + z)
    {+.1}
= y + z
    {+.1}
= (Zero + y) + z
```

Caso indutivo

Hipótese: `x + (y + z) = (x + y) + z`

Tese: `Succ x + (y + z) = (Succ x + y) + z`

```
    Succ x + (y + z)
      {+.2}
  = Succ (x + (y + z))
      {hipótese de indução}
  = Succ ((x + y) + z)
      {+.2}
  = Succ (x + y) + z
      {+.2}
  = (Succ x + y) + z
```

Dois lemas auxiliares

Distributividade de `reverse` sobre `++`

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

Atenção à inversão da ordem dos argumentos!

Para provar o lema acima, necessitamos de mostrar:

Associatividade de ++

```
(xs ++ ys) ++ zs = xs ++ (ys ++ zs)
```

Exercício: provar estes lemas usando indução.

Associatividade de ++
Caso base

```
  [] ++ ( ys ++ zs)
    {++.1}
= ys + zs
    {++.1}
= ([] ++ ys) + zs
```

Caso indutivo

Hipótese: `xs ++ (ys ++ zs) = (xs ++ ys) ++ zs`

Tese: `(x:xs) ++ (ys ++ zs) = ((x:xs) ++ ys) ++ zs`

```
  (x:xs) ++ (ys ++ zs)
    {++.2}
= x:(xs ++ (ys ++ zs))
    {hipótese de indução}
= x:((xs ++ ys) ++ zs)
    {++.2}
= x:(xs ++ ys) ++ zs
    {++.2}
= ((x:xs) ++ ys) ++ zs
```

Distributividade de `reverse` sobre ++
Caso base

```
  reverse ([] ++ ys)
    {++.1}
= reverse ys
    Elemento neutro de ++ (demonstrado acima)
= reverse ys ++ []
    {reverse.1}
= reverse ys ++ reverse []
```

Caso indutivo

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

Tese: `reverse ((x:xs) ++ ys) = reverse ys ++ reverse (x:xs)`

```
  reverse ((x:xs) ++ ys)
    {++.2}
= reverse (x:(xs ++ ys))
    {reverse.2}
= reverse (xs ++ ys) ++ [x]
    {hipótese de indução}
= (reverse ys ++ reverse xs) ++ [x]
    {Associatividade de ++}
= reverse ys ++ (reverse xs ++ [x])
    {reverse.2}
= reverse ys ++ reverse (x:xs)
```

Exercício 91: `map f (map g xs) = map (f ∘ g) xs`
Caso base

```
    map f (map g [])
    {map.1}
=   map f []
    {map.1}
=   []
    {map.1}
=   map (f ∘ g) []
```

Caso indutivo

Hipótese: `map f (map g xs) = map (f ∘ g) xs`

Tese: `map f (map g (x:xs)) = map (f ∘ g) (x:xs)`

```
  map f (map g (x:xs))
    {map.2}
= map f (g x):(map g xs)
    {map.2}
```

```
= (f (g x)):(map f (map g xs))
     {hipótese de indução}
= (f (g x)):(map (f ∘ g) xs)
     {definição de ∘}
= ((f ∘ g) x):(map (f ∘ g) xs)
     {map.2}
= map (f ∘ g) (x:xs)
```

Exercício 92: `take n xs ++ drop n xs = xs`
Caso base: n = 0

```
   take 0 xs ++ drop 0 xs
    {take.2}
=  [] ++ drop 0 xs
    {drop.2}
=  [] ++ xs
    {++.1}
=  []
```

Caso indutivo

Hipótese: `take n xs ++ drop n xs = xs`

Tese: `take (n+1) xs ++ drop (n+1) xs = xs`

Temos 2 casos:

1. `xs = []`:

```
   take (n+1) [] ++ drop (n+1) []
    {take.1}
=  [] ++ drop (n+1) []
    {drop.1}
=  [] ++ []
    {++.1}
=  []
```

2. `xs = (y:ys)`:

```
  take (n+1) (y:ys) ++ drop (n+1) (y:ys)
    {take.3}
= y:(take n ys) ++ drop (n+1) (y:ys)
    {drop.3}
= y:(take n ys) ++ drop n ys
    {++.2}
= y:(take n ys ++ drop n ys)
    {hipótese de indução}
= y:ys
```

Exercício 98: `folhas t = 1 + nos t`

Começamos por definir as funções `folhas` e `nos`:

```
folhas :: Arv a -> Int
folhas Folha  = 1                                    (folhas.1)
folhas (No x esq dir) = folhas esq + folhas dir      (folhas.2)
```

```
nos :: Arv a -> Int
nos Folha  = 0                                       (nos.1)
nos (No x esq dir) = 1 + nos esq + nos dir           (nos.2)
```

Caso base

```
   folhas Folha
    {folhas.1}
=  1
    {elemento neutro da adição}
=  1 + 0
    {nos.1}
=   1 + nos Folha
```

Caso indutivo

```
    folhas (No x esq dir)
       {folhas.2}
  = folhas esq + folhas dir
       {hipótese de indução sobre esq e dir}
  = 1 + nos esq + 1 + nos dir
       {comutatividade e associatividade de +}
  = 1 + (1 + nos esq + nos dir)
       {nos.2}
  = 1 + nos (No x esq dir)
```