Correção de programas: Listas e Árvores

Programação Funcional

Baseado nos slides do Prof. Rodrigo Ribeiro

Objetivos

Objetivos

- Construção de provas por indução envolvendo algoritmos sobre listas e árvores em Haskell.

Indução sobre listas

Indução sobre listas

- Para provar uma propriedade

```
forall xs :: [a] . P (xs)
```

- Devemos provar:
  - P([])
  - forall x xs. P(xs) -> P (x : xs)

Indução sobre listas

- Provar a seguinte propriedade

```
forall xs ys. length (xs ++ ys) = length xs + length ys
```

Indução sobre listas

- Lembrando:

```haskell
(++) :: [a] -> [a] -> [a]
[] ++ ys = ys
(x : xs) ++ ys = x : (xs ++ ys)

length :: [a] -> Int
length [] = 0
length (_ : xs) = 1 + length xs
```

Indução sobre listas

- Caso base ($xs = []$). Suponha ys arbitrário.

```
length ([] ++ ys) = -- def. de ++
```

Indução sobre listas

- Caso base ($xs = []$). Suponha ys arbitrário.

```
length ([] ++ ys) = -- def. de ++
length ys         = -- aritmética
```

Indução sobre listas

- Caso base ($xs = []$). Suponha ys arbitrário.

```
length ([] ++ ys) = -- def. de ++
length ys         = -- aritmética
0 + length ys     = -- def. de length
```

Indução sobre listas

- Caso base (xs = []). Suponha ys arbitrário.

```
length ([] ++ ys) = -- def. de ++
length ys         = -- aritmética
0 + length ys     = -- def. de length
length [] + length ys
```

Indução sobre listas

- Caso base (xs = z : zs). Suponha z, zs e ys arbitrários.
  - H.I. length (zs ++ ys) = length zs + length ys.

```
length ((z : zs) ++ ys)      = -- def. de ++
```

Indução sobre listas

- Caso base (xs = z : zs). Suponha z, zs e ys arbitrários.
  - H.I. length (zs ++ ys) = length zs + length ys.

```
length ((z : zs) ++ ys)      = -- def. de ++
length (z : (zs ++ ys))      = -- def. de length
```

Indução sobre listas

- Caso base ($xs = z : zs$). Suponha z, zs e ys arbitrários.
  - H.I. $\text{length } (zs ++ ys) = \text{length } zs + \text{length } ys$.

```
length ((z : zs) ++ ys)     = -- def. de ++
length (z : (zs ++ ys))     = -- def. de length
1 + length (zs ++ ys)       = -- H.I.
```

Indução sobre listas

- Caso base (xs = z : zs). Suponha z, zs e ys arbitrários.
  - H.I. length (zs ++ ys) = length zs + length ys.

```
length ((z : zs) ++ ys)      = -- def. de ++
length (z : (zs ++ ys))      = -- def. de length
1 + length (zs ++ ys)        = -- H.I.
1 + (length zs + length ys) = -- assoc. da soma
```

Indução sobre listas

- Caso base ($xs = z : zs$). Suponha z, zs e ys arbitrários.
  - H.I. length (zs ++ ys) = length zs + length ys.

```
length ((z : zs) ++ ys)       = -- def. de ++
length (z : (zs ++ ys))       = -- def. de length
1 + length (zs ++ ys)         = -- H.I.
1 + (length zs + length ys)   = -- assoc. da soma
(1 + length zs) + length ys   = -- def. de length
```

Indução sobre listas

- Caso base (xs = z : zs). Suponha z, zs e ys arbitrários.
  - H.I. length (zs ++ ys) = length zs + length ys.

```
length ((z : zs) ++ ys)      = -- def. de ++
length (z : (zs ++ ys))      = -- def. de length
1 + length (zs ++ ys)        = -- H.I.
1 + (length zs + length ys) = -- assoc. da soma
(1 + length zs) + length ys = -- def. de length
length (z : zs) + length ys
```

Indução sobre listas

- Provar a seguinte propriedade de map:

```
forall xs :: [a]. map id xs = xs
```

Indução sobre listas

- Caso base (xs = [])

```
map id [] = -- def. de map
[]
```

Indução sobre listas

- Caso xs = y : ys. Suponha y e ys arbitrários.
  - H.I. map id ys = ys.

```
map id (y : ys)  = -- def. de map
```

Indução sobre listas

- Caso xs = y : ys. Suponha y e ys arbitrários.
  - H.I. map id ys = ys.

```
map id (y : ys)  = -- def. de map
id y : map id ys = -- H.I.
```

Indução sobre listas

- Caso $xs = y : ys$. Suponha y e ys arbitrários.
  - H.I. map id ys = ys.

```
map id (y : ys)  = -- def. de map
id y : map id ys = -- H.I.
id y : ys        = -- def. de id
```

Indução sobre listas

- Caso xs = y : ys. Suponha y e ys arbitrários.
  - H.I. map id ys = ys.

```
map id (y : ys)  = -- def. de map
id y : map id ys = -- H.I.
id y : ys        = -- def. de id
y : ys
```

Map fusion

Map fusion

- Teorema que permite compor dois caminhamentos sobre uma lista como um único.
- Formalmente

```
forall xs :: [a], f :: a -> b, g :: b -> c.
    (map g . map f) xs = map (g . f) xs
```

Map fusion

- Caso base (xs $=$ []). Suponha f e g arbitrários.

```
(map g . map f) [] = -- def. de (.)
```

Map fusion

- Caso base ($xs = []$). Suponha f e g arbitrários.

```
(map g . map f) [] = -- def. de (.)
map g (map f [])   = -- def. de map
```

Map fusion

- Caso base (xs = []). Suponha f e g arbitrários.

```
(map g . map f) [] = -- def. de (.)
map g (map f [])   = -- def. de map
map g []           = -- def. de map
```

Map fusion

- Caso base (xs = []). Suponha f e g arbitrários.

```
(map g . map f) [] = -- def. de (.)
map g (map f [])   = -- def. de map
map g []           = -- def. de map
[]                 = -- def. de map
```

Map fusion

- Caso base ($xs = []$). Suponha f e g arbitrários.

```
(map g . map f) [] = -- def. de (.)
map g (map f [])   = -- def. de map
map g []           = -- def. de map
[]                 = -- def. de map
map (g . f) []
```

Map fusion

- Caso recursivo (xs = y : ys). Suponha f, g, y, ys arbitrários.
  - H.I. (map g . map f) ys = map (g . f) ys

```
(map g . map f) (y : ys)          = -- def. de (.)
```

Map fusion

- Caso recursivo (xs = y : ys). Suponha f, g, y, ys arbitrários.
  - H.I. (map g . map f) ys = map (g . f) ys

```
(map g . map f) (y : ys)          = -- def. de (.)
map g (map f (y : ys))            = -- def. de map
```

Map fusion

- Caso recursivo (xs = y : ys). Suponha f, g, y, ys arbitrários.
  - H.I. (map g . map f) ys = map (g . f) ys

```
(map g . map f) (y : ys)          = -- def. de (.)
map g (map f (y : ys))            = -- def. de map
map g (f y : map f ys)            = -- def. de map
```

Map fusion

- Caso recursivo (xs = y : ys). Suponha f, g, y, ys arbitrários.
  - H.I. (map g . map f) ys = map (g . f) ys

```
(map g . map f) (y : ys)          = -- def. de (.)
map g (map f (y : ys))            = -- def. de map
map g (f y : map f ys)            = -- def. de map
g (f y) : (map g (map f ys))      = -- def. de (.)
```

Map fusion

- Caso recursivo (xs = y : ys). Suponha f, g, y, ys arbitrários.
  - H.I. (map g . map f) ys = map (g . f) ys

```
(map g . map f) (y : ys)           = -- def. de (.)
map g (map f (y : ys))             = -- def. de map
map g (f y : map f ys)             = -- def. de map
g (f y) : (map g (map f ys))       = -- def. de (.)
((g . f) y) : (map g (map f ys))   = -- def. de (.)
```

Map fusion

- Caso recursivo (xs = y : ys). Suponha f, g, y, ys arbitrários.
  - H.I. (map g . map f) ys = map (g . f) ys

```
(map g . map f) (y : ys)         = -- def. de (.)
map g (map f (y : ys))           = -- def. de map
map g (f y : map f ys)           = -- def. de map
g (f y) : (map g (map f ys))     = -- def. de (.)
((g . f) y) : (map g (map f ys)) = -- def. de (.)
((g . f) y) : ((map g . map f) ys) = -- H.I.
```

Map fusion

- Caso recursivo (xs = y : ys). Suponha f, g, y, ys arbitrários.
  - H.I. (map g . map f) ys = map (g . f) ys

```
(map g . map f) (y : ys)              = -- def. de (.)
map g (map f (y : ys))                = -- def. de map
map g (f y : map f ys)                = -- def. de map
g (f y) : (map g (map f ys))          = -- def. de (.)
((g . f) y) : (map g (map f ys))      = -- def. de (.)
((g . f) y) : ((map g . map f) ys)    = -- H.I.
((g . f) y) : map (g . f) ys          = -- def. de map
```

Map fusion

- Caso recursivo (xs = y : ys). Suponha f, g, y, ys arbitrários.
  - H.I. (map g . map f) ys = map (g . f) ys

```
(map g . map f) (y : ys)           = -- def. de (.)
map g (map f (y : ys))             = -- def. de map
map g (f y : map f ys)             = -- def. de map
g (f y) : (map g (map f ys))       = -- def. de (.)
((g . f) y) : (map g (map f ys))   = -- def. de (.)
((g . f) y) : ((map g . map f) ys) = -- H.I.
((g . f) y) : map (g . f) ys       = -- def. de map
map (g . f) (y : ys)
```

Reverse

Reverse

- Provar a seguinte propriedade:

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

Reverse

- Caso base ($xs = []$). Suponha ys arbitrário.

```
reverse ([] ++ ys) = -- def. de ++
```

Reverse

- Caso base ($xs = []$). Suponha ys arbitrário.

```
reverse ([] ++ ys) = -- def. de ++
reverse ys         = -- Prop. forall ys. ys ++ [] = ys
```

Reverse

- Caso base (xs = []). Suponha ys arbitrário.

```
reverse ([] ++ ys) = -- def. de ++
reverse ys         = -- Prop. forall ys. ys ++ [] = ys
reverse ys ++ []   =
```

Reverse

- Caso base (xs = []). Suponha ys arbitrário.

```
reverse ([] ++ ys) = -- def. de ++
reverse ys         = -- Prop. forall ys. ys ++ [] = ys
reverse ys ++ []   = -- def. reverse
reverse ys ++ reverse []
```

Reverse

- Caso recursivo (xs = z : zs). Suponha z, zs e ys arbitrários.
  - H.I. ~reverse (zs + ys) = reverse ys ++ reverse zs~.

```
reverse ((z : zs) ++ ys)            = -- def. de ++
```

Reverse

- Caso recursivo (xs = z : zs). Suponha z, zs e ys arbitrários.
  - H.I. ~reverse (zs + ys) = reverse ys ++ reverse zs~.

```
reverse ((z : zs) ++ ys)          = -- def. de ++
reverse (z : (zs ++ ys))          = -- def. de reverse
```

Reverse

- Caso recursivo ($xs = z : zs$). Suponha z, zs e ys arbitrários.
  - H.I. ~reverse (zs + ys) = reverse ys ++ reverse zs~.

```
reverse ((z : zs) ++ ys)          = -- def. de ++
reverse (z : (zs ++ ys))          = -- def. de reverse
reverse (zs ++ ys) ++ [z]         = -- H.I.
```

Reverse

- Caso recursivo ($xs = z : zs$). Suponha z, zs e ys arbitrários.
  - H.I. ~reverse (zs + ys) = reverse ys ++ reverse zs~.

```
reverse ((z : zs) ++ ys)             = -- def. de ++
reverse (z : (zs ++ ys))             = -- def. de reverse
reverse (zs ++ ys) ++ [z]            = -- H.I.
(reverse ys ++ reverse zs) ++ [z] = -- Prop. ++ assoc.
```

Reverse

- Caso recursivo (xs = z : zs). Suponha z, zs e ys arbitrários.
  - H.I. ~reverse (zs + ys) = reverse ys ++ reverse zs~.

```
reverse ((z : zs) ++ ys)           = -- def. de ++
reverse (z : (zs ++ ys))           = -- def. de reverse
reverse (zs ++ ys) ++ [z]          = -- H.I.
(reverse ys ++ reverse zs) ++ [z] = -- Prop. ++ assoc.
reverse ys ++ (reverse zs ++ [z]) = -- def. de reverse
```

Reverse

- Caso recursivo (xs = z : zs). Suponha z, zs e ys arbitrários.
  - H.I. ~reverse (zs + ys) = reverse ys ++ reverse zs~.

```
reverse ((z : zs) ++ ys)            = -- def. de ++
reverse (z : (zs ++ ys))            = -- def. de reverse
reverse (zs ++ ys) ++ [z]           = -- H.I.
(reverse ys ++ reverse zs) ++ [z] = -- Prop. ++ assoc.
reverse ys ++ (reverse zs ++ [z]) = -- def. de reverse
reverse ys ++ (reverse (z : zs))
```

Fold-map fusion

Fold-map fusion

- Permite combinar duas operações sobre listas em uma única.
  - Idéia subjacente ao framework map/reduce.

```
forall xs f g v.
  (foldr g v . map f) xs = foldr (g . f) v xs
```

Fold-map fusion

- Caso base (xs = []). Suponha f, g e v arbitrários.

```
(foldr g v . map f) [] = -- def. de (.)
```

Fold-map fusion

- Caso base ($xs = []$). Suponha f, g e v arbitrários.

```
(foldr g v . map f) [] = -- def. de (.)
foldr g v (map f [])   = -- def. de map
```

Fold-map fusion

- Caso base ($xs = []$). Suponha f, g e v arbitrários.

```
(foldr g v . map f) [] = -- def. de (.)
foldr g v (map f [])   = -- def. de map
foldr g v []           = -- def. de foldr
```

Fold-map fusion

- Caso base ($xs = []$). Suponha f, g e v arbitrários.

```
(foldr g v . map f) [] = -- def. de (.)
foldr g v (map f [])   = -- def. de map
foldr g v []           = -- def. de foldr
v                      = -- def. de foldr
```

Fold-map fusion

- Caso base (xs = []). Suponha f, g e v arbitrários.

```
(foldr g v . map f) [] = -- def. de (.)
foldr g v (map f [])   = -- def. de map
foldr g v []           = -- def. de foldr
v                      = -- def. de foldr
foldr (g . f) v []
```

Fold-map fusion

- Caso indutivo (xs = y : ys). Suponha f, g, v, y e ys arbitrários.
  - H.I. (foldr g v . map f) ys = foldr (g . f) v ys.

```
(foldr g v . map f) (y : ys)        = -- def. de (.)
```

Fold-map fusion

- Caso indutivo (xs = y : ys). Suponha f, g, v, y e ys arbitrários.
  - H.I. (foldr g v . map f) ys = foldr (g . f) v ys.

```
(foldr g v . map f) (y : ys)        = -- def. de (.)
foldr g v (map f (y : ys))          = -- def. de map
```

Fold-map fusion

- Caso indutivo (xs = y : ys). Suponha f, g, v, y e ys arbitrários.
  - H.I. (foldr g v . map f) ys = foldr (g . f) v ys.

```
(foldr g v . map f) (y : ys)        = -- def. de (.)
foldr g v (map f (y : ys))          = -- def. de map
foldr g v (f y : map f ys)          = -- def. de foldr
```

Fold-map fusion

- Caso indutivo (xs = y : ys). Suponha f, g, v, y e ys arbitrários.
  - H.I. (foldr g v . map f) ys = foldr (g . f) v ys.

```
(foldr g v . map f) (y : ys)        = -- def. de (.)
foldr g v (map f (y : ys))          = -- def. de map
foldr g v (f y : map f ys)          = -- def. de foldr
g (f y) (foldr g v (map f ys))      = -- def. de (.)
```

Fold-map fusion

- Caso indutivo (xs = y : ys). Suponha f, g, v, y e ys arbitrários.
  - H.I. (foldr g v . map f) ys = foldr (g . f) v ys.

```
(foldr g v . map f) (y : ys)       = -- def. de (.)
foldr g v (map f (y : ys))         = -- def. de map
foldr g v (f y : map f ys)         = -- def. de foldr
g (f y) (foldr g v (map f ys))     = -- def. de (.)
(g . f) y ((foldr g v . map f) ys) = -- H.I.
```

Fold-map fusion

- Caso indutivo (xs = y : ys). Suponha f, g, v, y e ys arbitrários.
  - H.I. (foldr g v . map f) ys = foldr (g . f) v ys.

```
(foldr g v . map f) (y : ys)        = -- def. de (.)
foldr g v (map f (y : ys))          = -- def. de map
foldr g v (f y : map f ys)          = -- def. de foldr
g (f y) (foldr g v (map f ys))      = -- def. de (.)
(g . f) y ((foldr g v . map f) ys) = -- H.I.
(g . f) y (foldr (g . f) v ys)      = -- def. de foldr
```

Fold-map fusion

- Caso indutivo (xs = y : ys). Suponha f, g, v, y e ys arbitrários.
  - H.I. (foldr g v . map f) ys = foldr (g . f) v ys.

```
(foldr g v . map f) (y : ys)       = -- def. de (.)
foldr g v (map f (y : ys))         = -- def. de map
foldr g v (f y : map f ys)         = -- def. de foldr
g (f y) (foldr g v (map f ys))     = -- def. de (.)
(g . f) y ((foldr g v . map f) ys) = -- H.I.
(g . f) y (foldr (g . f) v ys)     = -- def. de foldr
foldr (g . f) v (y : ys)
```

Árvores

Árvores

- Definição de árvores binárias

```
data Tree a
    = Empty
    | Node a (Tree a) (Tree a)
      deriving (Eq, Ord, Show)
```

Árvores

- Para provar propriedades sobre árvores binárias, basta provar:
  - P(Empty)
  - forall l r x. P(l) -> P(r) -> P(Node x l r)

Árvores

- Algumas funções

```haskell
size :: Tree a -> Int
size Empty = 0
size (Node _ l r) = 1 + size l + size r

height :: Tree a -> Int
height Empty = 0
height (Node _ l r) = 1 + max (height l) (height r)
```

Árvores

- Provar que:

```
forall t. height t <= size t
```

Árvores

- Caso base (t = Empty):

```
height Empty  = -- def. height
```

Árvores

- Caso base (t = Empty):

```
height Empty  = -- def. height
0            <= -- aritmética
```

Árvores

- Caso base ($t = Empty$):

```
height Empty  = -- def. height
0            <= -- aritmética
0             = -- def. size
```

Árvores

- Caso base (t = Empty):

```
height Empty  = -- def. height
0            <= -- aritmética
0            =
size Empty
```

Árvores

- Caso recursivo: (t = Node x l r).
  - HI1. height l <= size l
  - HI2. height r <= size r.

```
height (Node x l r)              = -- def. de height
```

Árvores

- Caso recursivo: (t = Node x l r).
  - HI1. height l <= size l
  - HI2. height r <= size r.

```
height (Node x l r)           = -- def. de height
1 + max (height l) (height r) <= -- H.I.
```

Árvores

- Caso recursivo: (t = Node x l r).
  - HI1. height l <= size l
  - HI2. height r <= size r.

```
height (Node x l r)              = -- def. de height
1 + max (height l) (height r) <= -- H.I.
1 + max (size l) (size r)     <= -- aritmética
```

Árvores

- Caso recursivo: (t = Node x l r).
  - HI1. height l <= size l
  - HI2. height r <= size r.

```
height (Node x l r)                 = -- def. de height
1 + max (height l) (height r) <= -- H.I.
1 + max (size l) (size r)     <= -- aritmética
1 + size l + size r           = -- def. de size
```

Árvores

- Caso recursivo: (t = Node x l r).
  - HI1. height l <= size l
  - HI2. height r <= size r.

```
height (Node x l r)               = -- def. de height
1 + max (height l) (height r) <= -- H.I.
1 + max (size l) (size r)         <= -- aritmética
1 + size l + size r               =
size (Node x l r)
```

Exercícios

Exercício

- Prove que a concatenação de listas é uma operação associativa, isto é:

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