Correção de programas.

Programação Funcional

Prof. Rodrigo Ribeiro

Setup

```haskell
module Main where

main :: IO ()
main = return ()
```

Última aula

- Vimos como a transparência referencial permite provarmos a correção de programas usando equações.
- Provas sobre programas recursivos envolvem indução matemática.

Nessa aula

- Vimos indução sobre programas envolvendo números naturais.
- Aplicaremos estratégias similares para programas envolvendo listas.

Indução

- Para provar uma propriedade

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

Devemos provar:

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

Exemplo

- Provar que:

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

Provaremos por indução sobre a lista `xs`.

Exemplo

- Definições de `length` e `(++)`:

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

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

Exemplo

- Caso base: `xs = []`. Suponha `ys :: [a]` arbitrário.

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

Exemplo

- Caso base: `xs = z : zs`. Suponha `z :: a`, `zs ys :: [a]` arbitrários e que `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) = -- aritmética
(1 + length zs) + length ys = -- def. de length
length (z : zs) + length ys
```

Exemplo

- Provar a seguinte propriedade de `map`:

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

Exemplo

- Caso `xs = []`

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

Exemplo

- Caso `xs = y : ys`. Suponha `y :: a` e `ys :: [a]` arbitrários e que `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

- 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 = []`.

```
(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 indutivo: `xs = y : ys`. Suponha `y` e `ys` arbitrários e que `(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

- Provar a seguinte propriedade:

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

Reverse

- Caso `xs = []`. Suponha `ys` arbitrário.

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

Reverse

- Caso `xs = z : zs`. Suponha `z`, `zs` e `ys` arbitrários e que `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

- 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 (.)
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 e que `(foldr g v . map f) ys = foldr (g . f) v ys`.

```haskell
(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

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

Exemplo

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

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

Exemplo

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

Exemplo

- Provar que:

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

Exemplo

- Caso base (t = `Leaf`):

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

Exemplo

- Caso recursivo: (t = `Node x l r`). Suponha que `height l <= size l` e `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)
```

Exercício

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

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

Exercício

- Prove a seguinte propriedade sobre `map`:

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

Exercício

- Prove a seguinte propriedade sobre `map`:

```
forall xs f. length (map f xs) = length xs
```

Exercício

- Prove a seguinte propriedade:

```
forall xs f. reverse (map f xs) = map f (reverse xs)
```

Exercício

- Considere a função `toList`:

```
toList :: Tree a -> [a]
toList Leaf = []
toList (Node x l r) = toList l ++ [x] ++ toList r
```

Exercício

- Considere a função `member`:

```haskell
member :: Eq a => a -> Tree a -> Bool
member _ Leaf = False
member v (Node x l r)
  = v == x || member v l || member v r
```

Prove que:

```haskell
forall t x. member x t = elem x (toList t)
```

Exercício

- Considere a função `elem`:

```
elem :: Eq a => a -> [a] -> Bool
elem x [] = False
elem x (y : ys) = x == y || elem x ys
```

Prove a propriedade:

```
forall xs ys x.
  elem x (xs ++ ys) = elem x xs || elem x ys
```