# Programação Funcional

(COMP0393)

Leila M. A. Silva

dcomp
departamento de computação

# Raciocinando sobre Programas

(COMP0393)

## Aula 10

dcomp departamento de computação

# Raciocinando em Programação Funcional

- Podemos provar que um dado programa possui uma propriedade para quaisquer que sejam os dados de entrada.
- Para efetuar estas provas podemos usar a indução matemática, assim como a utilizamos para elaborar estratégias de solução de problemas.

# Raciocinando em Programação Funcional

- As definições das funções podem ser vistas como descrições do comportamento destas funções. Por exemplo, considere as definições de `length`:

```
length [] = 0                          (eq.1)
length (x:xs) = 1 + length xs          (eq.2)
```

A partir destas definições podemos entender como `length` se comporta:

- A (eq.1) nos diz que quando a lista for vazia seu comprimento é zero.
- A (eq.2) nos diz que independente do valor particular assumido por `x` e `xs`, o tamanho da lista (`x:xs`) é o tamanho da lista `xs` acrescido de `1`.

# Raciocinando em Programação Funcional

- Desta forma, como sabemos qual o comportamento de `length` para uma lista não vazia, podemos afirmar que:

```
length [x] = 1
```

Por que podemos garantir isto? Porque podemos **provar** este fato a partir das definições das funções!

```
length [x]
    = length (x:[])          <usando a def de [x]>
    = 1 + length []          <usando a eq.2 de length>
    = 1 + 0                  <usando a eq.1 de length>
    = 1                      <usando álgebra>
```

# Raciocinando em Programação Funcional

- Podemos usar este mecanismo para provar propriedades mais gerais, como:

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

Neste caso preciso das definições de `length` e também de `++`

# Raciocinando em Programação Funcional

```
length [] = 0                          (eq.1)
length (x:xs) = 1 + length xs          (eq.2)
[] ++ zs = zs                          (eq.3)
(w:ws) ++ zs = w:(ws++zs)              (eq.4)
```

Desejo provar que: `length (xs ++ ys) = length xs + length ys`

Caso base : `xs=[]`

Lado esquerdo:

```
length (xs ++ ys)              <xs=[]>
   = length([] ++ ys)          <eq.3>
   = length ys
```

Lado direito:

```
length xs + length ys
     = length [] + length ys   <xs=[]>
     = 0 + length ys           <eq.1>
     = length ys               <álgebra>
```

Logo, no caso base a propriedade vale `length (xs ++ ys) =length xs + length ys`

# Raciocinando em Programação Funcional

```
length [] = 0                       (eq.1)
length (x:xs) = 1 + length xs       (eq.2)
[] ++ zs = zs                       (eq.3)
(w:ws) ++ zs = w:(ws++zs)           (eq.4)
```

HI: Suponho que a propriedade vale para uma lista `xs` com *k* elementos. Ou seja,

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

Caso geral: Desejo provar que a propriedade vale para uma lista `x:xs` de *k+1* elementos. Ou seja, desejo provar que `length ((x:xs) ++ ys) = length (x:xs) + length ys`

Lado esquerdo:

```
length ((x:xs) ++ ys)              <eq.4>
   = length(x:(xs++ ys))           <eq.2>
   = 1 + length (xs ++ ys)         <HI>
   = 1 + length xs + length ys
```

Lado direito:

```
length (x:xs) + length ys
    = 1 + length xs + length ys   <eq.2>
```

Logo, o lado esquerdo é igual ao direito e a propriedade vale para o caso geral. ∎

# Mais exemplos...

Suponha as seguintes definições de funções:

```
dobroLista [] = []                          (dL.1)
dobroLista (w:ws) = 2*w: dobroLista ws      (dL.2)
sum []  = 0                                 (s.1)
sum (y:ys) = y + sum ys                     (s.2)
```

Desejo provar a seguinte propriedade:

```
sum (dobroLista xs) = 2 * sum xs
```

Caso base: `xs=[]`

Lado esquerdo:

```
sum (dobroLista xs)          <xs=[]>
 = sum (dobroLista [])       <dL.1>
 = sum []                    <s.1>
 = 0
```

Lado direito:

```
2 * sum xs                   <xs=[]>
 = 2 * sum []                <s.1>
 = 2 * 0                     <álgebra>
 = 0                          ∎
```

# Mais exemplos...

Suponha as seguintes definições de funções:

```
dobroLista [] = []                          (dL.1)
dobroLista (w:ws) = 2*w: dobroLista ws      (dL.2)
sum []  = 0                                 (s.1)
sum (y:ys) = y + sum ys                     (s.2)
```

HI: A propriedade vale para uma lista `xs` de *k* elementos. Ou seja,

```
sum (dobroLista xs) = 2 * sum xs
```

Caso geral: Desejo provar a propriedade para uma lista `x:xs` de *k+1* elementos. Ou seja, desejo provar que `sum (dobroLista (x:xs)) = 2 * sum (x:xs)`

```
sum (dobroLista (x:xs))          <dL.2>
 = sum (2*x: dobroLista xs)      <s.2>
 = 2* x + sum (dobroLista xs)    <HI>
 = 2*x + 2 * sum xs              <álgebra>
 = 2* (x + sum xs)               <s.2>
 = 2 * sum (x:xs)                ∎
```

# Mais exemplos...

Suponha as seguintes definições de funções:

```
reverse [] = []                          (r.1)
reverse (w:ws) = reverse ws ++ [w]       (r.2)
[] ++ zs = zs                            (++.1)
(w:ws) ++ zs = w:(ws++zs)                (++.2)
```

Desejo provar a seguinte propriedade:

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

Caso base: `xs=[]`

Lado esquerdo:

```
reverse (xs ++ ys )          <xs=[]>
 = reverse ([] ++ ys)        <++.1>
 = reverse ys
```

Lado direito:

```
reverse ys ++ reverse xs     <xs=[]>
 = reverse ys ++ []          <lema.1>
 = reverse ys                ∎
```

Lema 1: Provar que `us ++ [] = us`

# Mais exemplos...

Suponha as seguintes definições de funções:

```
reverse []     = []                    (r.1)
reverse (w:ws) = reverse ws ++ [w]     (r.2)
[] ++ zs       = zs                    (++.1)
(w:ws) ++ zs   = w:(ws++zs)            (++.2)
```

HI: A propriedade vale para uma lista `xs` de *k* elementos. Ou seja,

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

Caso geral: Desejo provar a propriedade para uma lista `x:xs` de *k+1* elementos. Ou seja, desejo provar que `reverse ((x:xs) ++ ys) = reverse ys ++ reverse (x:xs)`

```
reverse ((x:xs) ++ ys)                  <++.2>
 = reverse (x: (xs++ys))                <r.2>
 = reverse (xs ++ ys) ++ [x]            <HI>
 = (reverse ys ++ reverse xs) ++ [x]    <++.assoc>
 = reverse ys ++ (reverse xs) ++ [x])   <r.2>
 = reverse ys ++ reverse (x:xs)          ∎
```

Lema 2 (++.assoc) : Provar que `(us ++ ws) ++ vs = us ++ (ws ++ vs)`

# Mais exemplos...

Suponha as seguintes definições de funções:

```
[] ++ zs = zs                          (++.1)
(w:ws) ++ zs = w:(ws++zs)              (++.2)
sum []  = 0                            (s.1)
sum (y:ys) = y + sum ys                (s.2)
```

Desejo provar a seguinte propriedade:

```
sum (xs ++ ys) = sum xs + sum ys
```

Caso base: `xs=[]`

Lado esquerdo:

```
sum ([] ++ ys)          <xs=[]>
 = sum ys               <++.1>
```

Lado direito:

```
sum xs + sum ys         <xs=[]>
 = sum [] + sum ys      <s.1>
 = 0  + sum ys          <álgebra>
 = sum ys               ∎
```

# Mais exemplos...

Suponha as seguintes definições de funções:

```
[] ++ zs = zs                    (++.1)
(w:ws) ++ zs = w:(ws++zs)        (++.2)
sum []  = 0                      (s.1)
sum (y:ys) = y + sum ys          (s.2)
```

HI: A propriedade vale para uma lista `xs` de *k* elementos. Ou seja,

```
sum (xs ++ ys) = sum xs + sum ys
```

Caso geral: Desejo provar a propriedade para uma lista `x:xs` de *k+1* elementos. Ou seja, desejo provar que `sum ((x:xs) ++ ys) = sum (x:xs) + sum ys`

```
sum ((x:xs) ++ ys)              <++.2>
 = sum (x: (xs++ ys))           <s.2>
 = x + sum (xs ++ ys)           <HI>
 = x + (sum xs + sum ys)        <assoc da soma>
 = (x + sum xs)+ sum ys         <s.2>
 = sum (x:xs) + sum ys          ∎
```

# Mais exemplos...

Suponha as seguintes definições de funções:

```
reverse [] = []                          (r.1)
reverse (w:ws) = reverse ws ++ [w]       (r.2)
[] ++ zs = zs                            (++.1)
(w:ws) ++ zs = w:(ws++zs)                (++.2)
```

Desejo provar a seguinte propriedade:

```
reverse (reverse(xs) = xs
```

Caso base: `xs=[]`

Lado esquerdo:

```
reverse (reverse xs)          <xs=[]>
 = reverse (reverse [])       <r.1>
 = reverse []                 <r.1>
 = []
```

Lado direito:

```
xs = []
```

# Mais exemplos...

Suponha as seguintes definições de funções:

```
reverse [] = []                          (r.1)
reverse (w:ws) = reverse ws ++ [w]       (r.2)
[] ++ zs = zs                            (++.1)
(w:ws) ++ zs = w:(ws++zs)                (++.2)
```

HI: A propriedade vale para uma lista `xs` de *k* elementos. Ou seja,

```
reverse (reverse xs) = xs
```

Caso geral: Desejo provar a propriedade para uma lista `x:xs` de *k+1* elementos. Ou seja, desejo provar que `reverse (reverse (x:xs)) = (x:xs)`

```
reverse(reverse (x:xs))                      <r.2>
 = reverse (reverse xs ++ [x])               <exemplo 3>
 = reverse [x] ++ reverse(reverse xs))       <HI>
 = reverse [x] ++ xs                         <def :>
 = reverse (x:[]) ++ xs                      <r.2>
 = (reverse [] ++ [x]) ++ xs                 <r.1>
 = ([] ++ [x]) ++ xs                         <++.1>
 = [x] ++ xs                                 <def :>
 = (x:[]) ++ xs                              <++.1>
 = x:([]++xs)                                <++.2>
 = x: xs                                     ∎
```

# Exercícios Recomendados

- Prove os lemas 1 e 2 enunciados anteriormente.
- Prove que as seguintes propriedades valem para todas listas finitas `xs`:

```
sum (reverse xs) = sum xs
product (reverse xs) = product xs
length (reverse xs) = length xs
```

- Prove que a seguinte propriedade vale para todas listas finitas de inteiros `xs` e `ys`:

```
elem z (xs ++ ys) = elem z xs || elem z ys
product (xs++ys) = (product xs) * (product ys)
```

- Para o exemplo dado na Aula 9 prove que:

```
refleteH (reflete H xss) = xss
refleteV (reflete V xss) = xss
```