Correção de programas.

Programação Funcional

Prof. Rodrigo Ribeiro

Setup

```haskell
module Main where

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

Matemática

- Funções matemática não dependem de valores “ocultos” ou que podem ser alterados.
  - Ex: $2 + 3 = 5$ tanto em 4 * (2 + 3) quanto em (2 + 3) * (2 + 3).
- Isso facilita a demonstração de propriedades sobre essas funções.

Matemática

- Exemplo de propriedades (teoremas):

$$\forall xy.x + y = y + x$$
$$\forall xy.x \times y = y \times x$$
$$\forall xyz.x + (y + z) = (x + y) + z$$
$$\forall x.x + 0 = 0 + x = x$$
$$\forall xyz.x \times (y + z) = (x \times y) + (x \times z)$$

Utilidade

- Teoremas podem ajudar na **performance**
  - Substituir implementações ineficientes por equivalentes mais eficientes.
- Teoremas são a forma de mostrar que seu código atende os requisitos corretamente.

Utilidade

"Correctness is clearly the prime quality. If a system does not do what it is supposed to do, then everything else about it matters little."

- Bertrand Meyer, criador da linguagem Eiffel.

Álgebra

- Em matemática, é comum termos demonstrações similares a:

$$
\begin{array}{lcl}
(a+b)^2 & = & \text{def. de } x^2 \\
(a+b)\times(a+b) & = & \textit{distr.} \\
((a+b)\times a)+((a+b)\times b) & = & \textit{comut.} \\
(a\times(a+b))+(b\times(a+b)) & = & \textit{distr.} \\
(a\times a+a\times b)+(b\times a+b\times b) & = & \ldots
\end{array}
$$

Álgebra

- Continuando...

$$
\begin{array}{lcl}
(a \times a + a \times b) + (b \times a + b \times b) & = & \textit{assoc.} \\
a \times a + (a \times b + b \times a) + b \times b & = & \textit{comut.} \\
a \times a + (a \times b + a \times b) + b \times b & = & \textit{comut.} \\
a^2 + 2 \times a \times b + b^2 & = & \textit{def. } dex^2 \textit{ e de}+
\end{array}
$$

Em Haskell

- Como Haskell possui transparência referencial, podemos provar propriedades sobre programas usando raciocínio baseado em equações, como na matemática.

Exemplo

- Considere a definição da função `reverse`:

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

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

Exemplo

- Provar que `forall x. reverse [x] = [x]`.

```
reverse [x]          = -- list notation
reverse (x : [])     = -- def. reverse
reverse [] ++ [x]    = -- def. reverse
[] ++ [x]            = -- def. ++
[x]
```

Análise de casos

- Em algumas situações, é necessário considerar as diferentes possibilidades de parâmetros de entrada.
- Exemplo: provar que `not` é involutivo.

```
forall x. not (not x) = x
```

Análise de casos

- Definição de `not`:

```
not :: Bool -> Bool
not False = True
not True  = False
```

Análise de casos

- Provando que `not (not x) = x`.
- Caso `x = False`:

```
not (not False) = -- def. de not
not True        = -- def. de not
False
```

Análise de casos

- Provando que `not (not x) = x` (cont.).
- Caso `x = True`:

```
not (not True) = -- def. de not
not False      = -- def. de not
True
```

Números naturais

- Representando números naturais na notação de Peano.

```
data Nat = Zero | Succ Nat
           deriving (Eq, Ord, Show)
```

- Exemplos

```
two :: Nat
two = Succ (Succ Zero)
```

Números naturais

- Representando a soma.

```
(.+.) :: Nat -> Nat -> Nat
Zero      .+. m = m -- 1
(Succ n') .+. m = Succ (n' .+. m) -- 2
```

Números naturais

- Exemplo:

```
(Succ (Succ Zero)) .+. (Succ Zero) = -- eq. 2
Succ ((Succ Zero) .+. (Succ Zero)) = -- eq. 2
Succ (Succ (Zero .+. (Succ Zero))) = -- eq. 1
Succ (Succ (Succ Zero))
```

Números naturais

- Usando a definição de soma (equação 1), temos que:

```
forall n. Zero .+. n = n
```

- Parece óbvio que a seguinte propriedade também deve ser verdadeira:

```
forall n. n .+. Zero = n
```

Números naturais

- Porém, a propriedade não é imediata a partir das equações 1 e 2 da adição.
- Afinal, não é possível determinar se `n = Zero` ou se `n = Succ n'`, para algum `n'` em

```
forall n. n .+. Zero = n
```

Números naturais

- Como a adição é definida recursivamente, não podemos usar análise de casos para concluir a prova de

```
forall n. n .+. Zero = n
```

- Para isso, devemos usar **indução**.

Tipos recursivos

- Provas envolvendo funções recursivas são realizadas por indução.
- Casos base são construtores do tipo que não envolvem recursão.
- Passo indutivo para construtores envolvendo recursão.

Indução sobre `Nat`

- Para provar `forall x :: Nat. P (x)`, basta provar:
  - `P(Zero)`.
  - `forall n :: Nat . P(n) -> P(Succ n)`.

Indução sobre `Nat`

- Para a propriedade

```
forall n. n .+. Zero = n
```

`P(n)` é dado por `n .+. Zero = n`.

Indução sobre `Nat`

- Para a propriedade

```
forall n. n .+. Zero = n
```

`P(Zero)` é dado por `Zero .+. Zero = Zero`.

Indução sobre `Nat`

- Para a propriedade

```
forall n. n .+. Zero = n
```

`forall n. P(n) -> P(Succ n)` é dado por:

```
forall n. n .+. Zero = n -> (Succ n) .+. Zero = (Succ n)
```

Indução sobre `Nat`

- Provando a propriedade

```
forall n. n .+. Zero = n
```

- Caso base: `n = Zero`.

```
Zero .+. Zero = -- def. de .+.
Zero
```

Indução sobre `Nat`

- Caso indutivo: `n = Succ n'`.
  - Hipótese de indução: `n' .+. Zero = n'`.

```
(Succ n') .+. Zero = -- def. de .+.
Succ (n' .+. Zero) = -- H.I.
Succ n'
```

Indução sobre `Nat`

- Mais um exemplo:

```
forall n m. Succ (n .+. m) = n .+. (Succ m)
```

- Prova por indução sobre `n :: Nat`.

Indução sobre `Nat`

- Caso base: `n = Zero`. Suponha `m :: Nat` arbitrário.

```
Succ (Zero .+. m) = -- def. de .+.
Succ m            = -- def. de .+.
Zero .+. Succ m
```

Indução sobre `Nat`

- Caso indutivo: `n = Succ n'`. Suponha `m :: Nat` arbitrário e que `Succ (n' .+. m) = n' .+. (Succ m)`. Temos:

```
Succ (Succ n') .+. m    = -- def. de .+.
Succ (Succ (n' .+. m)) = -- H.I.
Succ (n' .+. (Succ m)) = -- def. de .+.
(Succ n') .+. (Succ m)
```

Exercício

- Prove que, para todo `x :: a` e `f :: a -> b`, `map f [x] = [f x]`, usando a definição de `map`.

```
map :: (a -> b) -> [a] -> [b]
map _ [] = []
map f (x : xs) = f x : map f xs
```

Exercício

- Prove que a operação de disjunção, `(||)`, atende as seguintes propriedades:

```
forall a b c. a || (b || c) = (a || b) || c
forall a. a || False = a
forall b. False || b = b
```

Exercícios

- Prove que a adição é uma operação associativa, isto é:

```
forall n m p. (n .+. m) .+. p = n .+. (m .+. p)
```