Programação Funcional — BCC222

Aulas 9

# Provas Formais

Lucília Camarão de Figueiredo
Departamento de Ciência da Computação
Universidade Federal de Ouro Preto

# Provas Formais

Porque fazer provas?

- Sistemas críticos em segurança
  (pilotos automáticos, transações bancárias via internet, provadores de teoremas)

Porque não apenas usar *QuickCheck*?

- Quais são os testes 'apropriados'?
- Quantos casos de teste são 'suficientes'?

# O que é uma prova?

Uma (possível) definição:

"Um argumento formal mostrando a verdade de uma proposição"

Uma definição mais útil seria talvez:

"Uma análise passo a passo de uma proposição levando a afirmações básicas (axiomas), onde cada passo é obviamente correto, e cada axioma é obviamente verdadeiro"

Quando uma coisa é 'óbvia'?

Isso é determinado 'socialmente'!

# Pitagoras de novo

```
isTriple a b c = a*a + b*b == c*c
leg1 x y = x*x - y*y
leg2 x y = 2 * x * y
hyp x y = x*x + y*y

prop_triple x y = isTriple (leg1 x y) (leg2 x y) (hyp x y)
```

# Desenvolvendo as definições

```
isTriple a b c = a*a + b*b == c*c
```

```
isTriple(x*x - y*y) (2 * x * y) (x*x + y*y)
```

# Desenvolvendo as definições

```
isTriple a b c = a*a + b*b == c*c
```

```
isTriple(x*x - y*y) (2 * x * y) (x*x + y*y)
```

```
(x*x - y*y) * (x*x - y*y) + (2 * x * y) * (2 * x * y)
                          == (x*x + y*y) * (x*x + y*y)
```

# Aritmética

```
isTriple a b c = a*a + b*b == c*c

isTriple(x*x - y*y) (2 * x * y) (x*x + y*y)

(x*x - y*y) * (x*x - y*y) + (2 * x * y) * (2 * x * y)
                          == (x*x + y*y) * (x*x + y*y)
```

Lei: `(a+b)*(c+d) == a*c + a*d + b*c + b*d`

# Aritmética

```
isTriple a b c = a*a + b*b == c*c

isTriple(x*x - y*y) (2 * x * y) (x*x + y*y)

(x*x - y*y) * (x*x - y*y) + (2 * x * y) * (2 * x * y)
                          == (x*x + y*y) * (x*x + y*y)

Lei:  (a+b)*(c+d) == a*c + a*d + b*c + b*d
x*x*x*x - 2*x*x*y*y + y*y*y*y + 4*x*x*y*y
                          == x*x*x*x + 2*x*x*y*y + y*y*y*y

x*x*x*x + 2*x*x*y*y + y*y*y*y
                          == x*x*x*x + 2*x*x*y*y + y*y*y*y
```

# Indução

Suponha que queremos provar alguma propriedade de listas.

- Não sabemos o comprimento de uma lista arbitrária
- Mas podemos supor que ela termina (é finita) (Algumas vezes não!)
- E sabemos como ela é:

  Uma lista ou é vazia, ou possui uma cabeça e uma cauda

# Indução Matemática

Como indução funciona:

Para provar que uma propriedade `p :: [Int] -> Bool` vale para qualquer lista, devemos mostrar que:

- `p []`
- se `p xs` então `p (x:xs)` (para qualquer `x :: Int`)

O primeiro passo é denominado *caso base*,

o segundo passo é o *passo indutivo*, e

a afirmação `p xs` é a *hipótese de indução*.

# Um exemplo fácil

Queremos provar que, para toda lista `xs`:

```
length xs >= 0
```

# Um exemplo fácil

Queremos provar que, para toda lista `xs`:

```
length xs >= 0
```

Caso base:

```
length [] >= 0
```

# Um exemplo fácil

Queremos provar que, para toda lista `xs`:

```
length xs >= 0
```

Caso base:

```
length [] >= 0
```

Substituindo a definição `(length [] = 0)`:

```
0 >= 0
```

# Um exemplo fácil

Queremos provar que, para toda lista `xs`:

```
length xs >= 0
```

Caso base:

```
length [] >= 0
```

Substituindo a definição (`length [] = 0`):

```
0 >= 0
```

Caso indutivo:

se `length xs >= 0` então `length (x:xs) >= 0`

# Um exemplo fácil

Queremos provar que, para toda lista `xs`:

```
length xs >= 0
```

Caso base:

```
length [] >= 0
```

Substituindo a definição `(length [] = 0)`:

```
0 >= 0
```

Caso indutivo:

se `length xs >= 0` então `length (x:xs) >= 0`

Substituindo a definição `(length (x:xs) = 1 + length xs)`:

se `length xs >= 0` então `length xs + 1 >= 0`

# Indução Matemática

Indução sobre números naturais:

Para provar que uma propriedade `p` vale para qualquer número natural, devemos mostrar que:

- `p 0` (caso base)
- se `p n` então `p (n+1)` (passo indutivo)

A asserção `p n` é a hipótese de indução.

# Exemplo

Queremos provar que:

```
even x || even (x+1)
```

# Exemplo

Queremos provar que:

```
even x || even (x+1)
```

Base case:

```
even 0 || even (0+1)
```

(obvious)

# Exemplo

Queremos provar que:

```
even x || even (x+1)
```

Base case:

```
even 0 || even (0+1)
```

(obvious)

Caso indutivo:

se `even x || even (x+1)`

então `even (x+1) || even ((x+1)+1)`

# Exemplo

Queremos provar que:

```
even x || even (x+1)
```

Base case:

```
even 0 || even (0+1)
```

(obvious)

Caso indutivo:

se `even x || even (x+1)`

então `even (x+1) || even ((x+1)+1)`

Possíveis casos:

1. Suponha `even x`
2. Suponha `even (x+1)`

# Exemplo

Queremos provar que:

```
even x || even (x+1)
```

Base case:

```
even 0 || even (0+1)
```

(obvious)

Caso indutivo:

se `even x || even (x+1)`

então `even (x+1) || even ((x+1)+1)`

Possíveis casos:

1. Suponha `even x`
   então (obviamente) `even (x+2)`
2. Suponha `even (x+1)`
   terminamos