# Aula PL #01

19 de setembro de 2023 18:06

$$\begin{cases} f\ x = 2 * x \\ g\ x = x + 1 \end{cases} \quad \begin{cases} f = \text{succ} \\ g\ x = 2 * x \end{cases}$$

```
ghci> f x = 2 * x
ghci> g x = x + 1
ghci> (f.g) 5
```

```
ghci> f = succ
ghci> g x = 2 * x
ghci> f.g 5
```

(f . g) . h = f . (g . h)

A <- f - B <- g <- C
A <- f.g - C

Ex.3

f . Id = f
(lei 72)
<=> para todo o x, (f.id) x = f x
(lei 73)
<=> para todo o x f(id(x) = f x
(lei 74)
<=> para todo o x f(x) = f(x)

(f.g).h=f.(g.h)
<=> f ( g ( h x ) ) = f ( g ( h x ) )

<=> para todo o x, ((f.g).h) x = (f.(g.h)) x
<=> para todo o x,

Store c = take 10 . Nub . (x:)
Store aceita outros tipos => polimorfismo

Regras

$(f \times g) \cdot (h \times k) = f \cdot h \times g \cdot h$ (72)

$\Leftrightarrow \forall x,y \mid (f \times g)\ (h\ x, k\ y) = (f\ (h\ x), g\ (k\ y))$

(73) e (78)

$\Leftrightarrow \forall x,y \mid (f\ (h\ x), g\ (k\ y)) = (f\ (h\ x), g\ (k\ y))$

$\Leftrightarrow$ TRUE

| | | |
|---|---|---|
| **Igualdade extensional** | $f = g \Leftrightarrow (\forall x :: f\ x = g\ x)$ | (72) |
| **Def-comp** | $(f \cdot g)\ x = f\ (g\ x)$ | (73) |
| **Def-id** | $id\ x = x$ | (74) |
| **Def-const** | $\underline{k}\ x = k$ | (75) |
| **Notação-$\lambda$** | $f\ a = b \equiv f = \lambda a \to b$ | (76) |
| **Def-split** | $\langle f, g \rangle\ x = (f\ x, g\ x)$ | (77) |
| **Def-$\times$** | $(f \times g)\ (a, b) = (f\ a, g\ b)$ | (78) |

↳ Do formulário

$\pi_1 \cdot (f \times g) = f \cdot \pi_1$

(72) $\Leftrightarrow \forall x,y \mid (\pi_1 \cdot (f \times g))\ (x,y) = (f \cdot \pi_1)\ (x,y)$

(73) $\Leftrightarrow \forall x,y \mid \pi_1\ ((f \times g)\ (x,y)) = f\ (\pi_1\ (x,y))$

(78) $\Leftrightarrow \forall x,y \mid \pi_1\ (f\ x, g\ y) = f\ x$

$\Leftrightarrow \forall x,y \mid f\ x = f\ x$

$\Leftrightarrow$ TRUE

```
length :: [a] -> Z
length [] = 0
length (x:xs) = 1 + length xs
```

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

```
(++) :: [a] -> [a] -> [a]
(++) [] l = l
(++) (h:t) l = h : (++) t l
```

(++) [1,2,3] [5,6] ⇓

[1] : ([2] : ([3] : [5,6]))

- CURRY

```
add :: Int -> Int -> Int
add x y = x + y

add 2 :: Int -> Int
add 2 3 :: Int
```

Series of curried functions

→ Curry is a process of converting a function that takes multiple arguments

→ Curry is a process of converting a function that takes multiple arguments into a series of functions, each taking a single argument

```
curry :: ((a, b) -> c) -> a -> b -> c
curry f x y = f (x, y)
```

Curry the add function explicitly

```
addTwo :: Int -> Int
addTwo = curry (+) 2
```

- UNCURRY

```
uncurry :: (a -> b -> c) -> (a, b) -> c
uncurry f (x, y) = f x y
```

```
uA :: (Int, Int) -> Int
uA = uA add
```

⇒ uA (2, 3)

```
-- Original function taking a tuple
addTuple :: (Int, Int) -> Int
addTuple (x, y) = x + y

-- Using uncurry to adapt it for curried arguments
addCurried :: Int -> Int -> Int
addCurried = uncurry addTuple
```

The uncurry function in Haskell is used to transform a curried function (a function that takes multiple arguments as a sequence of single-argument functions) into a function that takes a tuple as its argument.

- Flip

```
flip :: (a -> b -> c) -> b -> a -> c
flip f x y = f y x
```