

# **Why functional programming matters**

**Jean Carlo Machado**

→ Title: Why functional programming matters

→ Author: Jhon Hughes, Glasgow

→ Year: 1990

# Sobre

CompuFácil

- php-src
  - git/git
- torvald/linux
  - vim/vim
  - Doctrine
- Zend Framework
  - phpunit

CODE WRITTEN IN HASKELL  
IS GUARANTEED TO HAVE  
NO SIDE EFFECTS.

...BECAUSE NO ONE  
WILL EVER RUN IT?



# História

# 1936 Imperativo

→ Alan Turing

→ computação universal

→ Receita

→ assembly

→ Goto

→ side effects

# 1950 Estruturado

- Agol
- while, for, if
- reuso
- sem goto

# Programação Funcional

~~mudança de estado~~

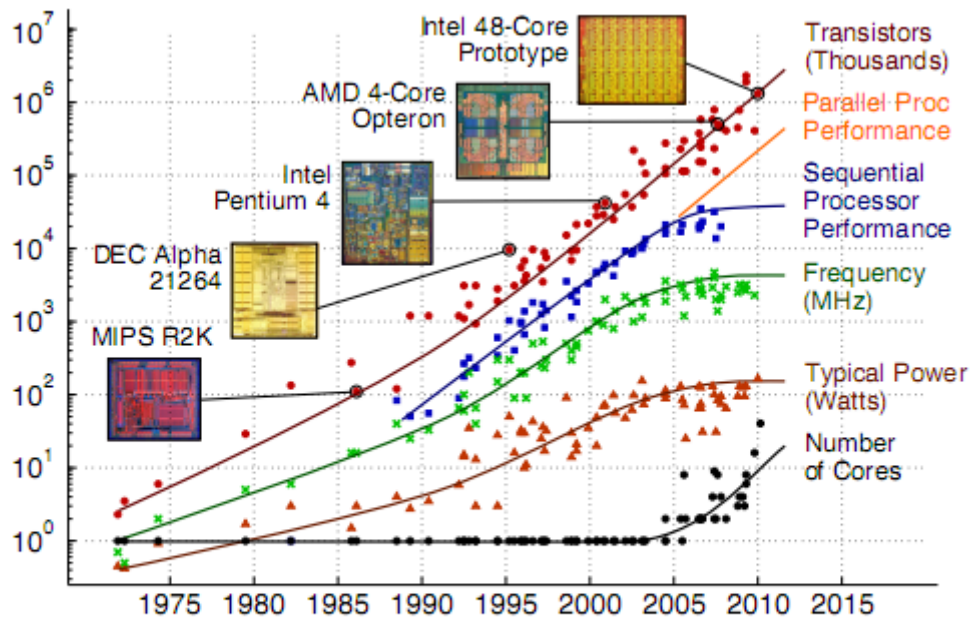


# Vantagens

- Facilita a modularização
- Concorrência nativa
- Desacelera o apodrecimento

# Modularização

- Mais que módulos
- Decompor os problemas em partes menores
- Re-compor com avaliação tardia e funções de alta ordem



Data partially collected by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond

Prepared by C. Batten - School of Electrical and Computer Engineering - Cornell University - 2005 - retrieved Dec 12 2012 -

<http://www.ece.cornell.edu/courses/ece5950/handouts/ece5950-overview.pdf>

# ~~Apodrecimento~~

- Impossível adicionar efeitos sem quebrar as interfaces
  - Quanto maior a interface mais feio o código
- Interfaces facilmente quebráveis com composição
  - Quebrar encoraja reuso
- Complexidade horizontal ao invés de vertical

# Prática

# Impureza == Estado

```
$inpureIncrement = function (&$count) {  
    $count++;  
};
```

# Pureza (relativa)

```
function pureIncrement($count) {  
    $myCounter = $count;  
    $inpureIncrement = function(&$myCounter){  
        $count++;  
    };  
    $inpureIncrement($myCounter);  
    return $myCounter;  
}
```

# Transparência referencial

```
let y = f x in y + y  
// substituível  
f x + f x
```



# Impuros por natureza

→ IO

→ print

→ random

# Funções de alta ordem

# Retornáveis

```
$sumPartial = function(int $a) {  
    return function(int $b) use ($a) {  
        return $a+b;  
    }  
};  
$sumOne = $sumPartial(1);  
$sumOne(7);  
//8  
$sumOne(5);  
//6
```

# Repassáveis

```
$incrementTwo = function ($n, $sumOne) {  
    return $sumOne($sumOne($n));  
};
```

# Aplicação parcial

```
$append3 = function($a, $b, $c) {  
    return [$a, $b, $c];  
};  
$append1And2 = \f\partial($append3)(1)(2);  
$append1And2(5)  
//[1,2,5]  
$append1And2(9)  
//[1,2,9]
```

# Fold

# Defina o básico

```
function sum($a, $b) {  
    return $a + $b;  
}  
//function product($a, $b);  
//function aORb($a, $b);  
//function append($a, $b);  
//function greater($a, $b);
```

```
function fold(  
    callable $f,  
    $init,  
    $list) {  
  
    if (empty($list)) {  
        return $init;  
    }  
    return $f(  
        fold($f, $init, allbutlast($list)),  
        last($list)  
    );  
};
```



# Componha

```
$sumList = fold('sum', 0);  
$sumList([1,2,3]);  
//6  
$multiplyList = fold('product', 0);  
$sumList([2,2,2]);  
//8
```

# Map

```
$double = partial('product')(2);  
$map($double, [1, 2, 4]);  
//[2,4,8]
```

```
$map = function($f, $list) {  
    $applyAndAppend=function($f,$list,$b){  
        return append($list, $f($b));  
    };  
    return fold($applyAndAppend,null)($list);  
}
```

# Árvores

```
$tree = [  
  1 => [  
    2,  
    3 => [  
      4,  
    ],  
  ],  
];  
foldTree($sum, $sum, 0, $tree)  
//10
```

# Map Tree

```
$tree = [  
    3 => [1,4],  
    1 => [1,5 => [1,2,3]]  
];  
$result = mapTree($double, $tree);  
//[  
//     6 => [2,8],  
//     2 => [2,10 => [2,4,6]]  
//];
```

**Avaliação tardia**

# **g (f input)**

- começa f só quando g tenta ler algo
- suspende f
- g roda até precisar ler mais dados

**Nth primes**



```
function isPrime($x) {  
    $seq=\f\takeFrom(  
        \f\infiniteSequence(2),  
        ($x-2)  
    );  
    $multipleOfA=  
        \f\partial('f\op\amultipleOfb')  
        ($x);  
    $result = \f\map($multipleOfA, $seq);  
    return \f\noneTrue($result);  
}
```

# Nth primes

```
function primes() {  
    foreach (\f\infiniteSequence(1) as $i) {  
        if (isPrime($i)) {  
            yield $i;  
        }  
    }  
}  
  
print_r(\f\takefrom(primes(), 10));  
//[1, 3, 5, 7, 11, 13, 17, 19, 23, 29]
```

# Conclusão

- Imperativo quando necessário
  - Funcional quando possível
    - Para novos tipos:
- defina as operações fundamentais
  - junte funções
- Seu trabalho não é resolver problemas
- Definir problemas de uma forma que eles se resolvam

# Referências

→ Why functional programming matters

→ <http://archive.li/uTYWZ#selection-407.912-407.919>

→ <https://medium.com/@jugoncalves/functional-programming-should-be-your-1-priority-for-2015-47dd4641d6b9>

→ <https://blog.inf.ed.ac.uk/sapm/2014/03/06/enemy-of-the-state-or-why-oop-is-not-suited-to-large-scale-software/>

→ <https://www.youtube.com/watch?v=7Zlp9rKHGD4>

→ <https://www.youtube.com/watch?v=q0HRCEKAcas>

# Dúvidas?

Github: <https://github.com/jeanCarloMachado>

Twitter: <https://twitter.com/JeanCarloMachad>

E-mail: [contato@jeancarlomachado.com.br](mailto:contato@jeancarlomachado.com.br)