Cálculo λ - O combinador Y

MCTA016-13 - Paradigmas de Programação

Emilio Francesquini
e.francesquini@ufabc.edu.br

2019.Q2

Centro de Matemática, Computação e Cognição
Universidade Federal do ABC

UFABC

Disclaimer

UFABC

- Estes slides foram preparados para o curso de **Paradigmas de Programação na UFABC**.
- Este material pode ser usado livremente desde que sejam mantidos, além deste aviso, os créditos aos autores e instituições.

O combinador Y

Y-Combinator

- Já fizemos algo nessa linha quando falamos do combinador Ω
- Basta misturarmos a ideia do Ω com uma função que queremos que seja ”carregada”junto
- **Haskell** Brooks **Curry** 😛 foi o primeiro a fazê-lo!
  - Haskell foi orientado de doutorado do David Hilbert

Wikipedia/Gleb.svechnikov CC BY-SA 4.0

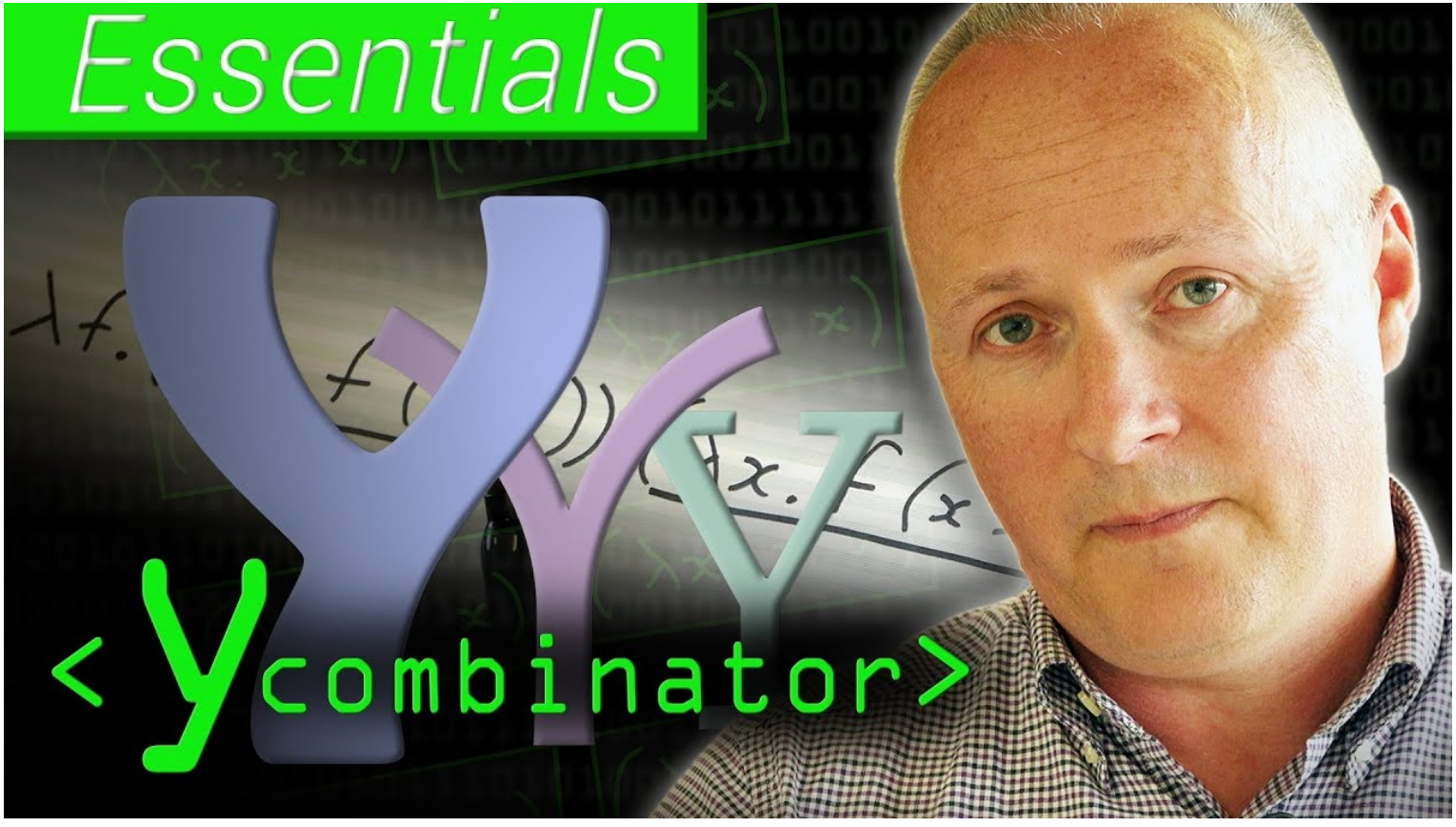
O combinador Y

UFABC

https://www.youtube.com/watch?v=BC8ZAMwfwi4

O combinador Y - Agora sim um link sério

UFABC



- Agora sim um link sério: *Essentials: Functional Programming's Y Combinator - Computerphile*
- `https://www.youtube.com/watch?v=9T8A89jgeTI`

Recursão - O exemplo do fatorial

UFABC

- Queremos escrever uma função fatorial recursiva
- Em Haskell, poderíamos escrevê-la da seguinte maneira:

```haskell
fat x = if x == 0 then 1 else x * fat (x - 1)
-- Ou usando lambda
fat = \x -> if x == 0 then 1 else x * fat (x - 1)
```

- Contudo, em cálculo λ isso não é possível!
  - Funções são anônimas! `fat` faz referência a si própria por nome!

UFABC

- Podemos contornar o problema criando uma nova versão de `fat` que não referencie nomes não definidos pela própria expressão

```haskell
fat = \f -> \x -> if x == 0 then 1 else x * f (x - 1)
-- Ou usando açúcar sintático
fat = \f x -> if x == 0 then 1 else x * f (x - 1)
```

Pergunta

Quero calcular fatorial de 3.

```haskell
fat ??? 3
```

O que passar como `f`?

Recursão - Fatorial

```haskell
fat = \f x -> if x == 0 then 1 else x * f (x - 1)
```

- O mais conveniente seria usar um valor para `f`, digamos `p`, tal que
  - `p` $\equiv$ `fat p`, ou em outras palavras, `p n` $\equiv$ `fat p n`
    - $3! \equiv$ `p 3` $\equiv$ `fat p 3` $\equiv 6$
  - Ou seja, queremos que `p` seja ponto fixo[1] da função `fat`.

[1] Um **ponto fixo** de uma função f é um elemento do domínio de uma função que é mapeado para si mesmo pela função. Por exemplo, $0, 1$ e -1 são pontos fixos de $f(x) = x^3$ pois $f(0) = 0^3 = 0$, $f(1) = 1^3 = 1$ e $f(-1) = -1^3 = -1$.

Ponto fixo de funções λ

UFABC

- Considere a expressão

```
1  A = (\x -> f (x x)) (\x -> f (x x))
```

  - Qualquer semelhança com o combinador Ω (`(\x -> x x) (\x -> x x)`) não é mera coincidência

- Logo:

```
1  A = (\x -> f (x x)) (\x -> f (x x))
2    = f ((\x -> f (x x)) (\x -> f (x x))) -- por
     ↪  redução β
3    = f A
```

- Ou seja, para todo `f`, **A é ponto fixo** de `f`

Ponto fixo de funções λ

UFABC

- Podemos, então, finalmente construir o combinador Y (*Y-combinator*)
- O combinador Y nada mais é do que uma expressão λ que devolve um ponto fixo para qualquer função λ

```
1  A = (\x -> f (x x)) (\x -> f (x x))
2  Y = \f -> A -- liga a variável f em A
3  Y = \f -> (\x -> f (x x)) (\x -> f (x x))
```

- Ou, em notação matemática, $\lambda f.\,(\lambda x.\,f\ (x\ x))\ \ (\lambda x.\,f\ (x\ x))$.
- Como que é? 

O combinador Y

UFABC

- O combinador Y nos permite escrever uma expressão λ cuja avaliação se repete indefinidamente, tal qual o combinador Ω, mas que diferentemente deste faz algo útil durante cada redução: a avaliação da função f
- O "pulo do gato"está no uso do ponto fixo de funções λ (que pode ser obtido pela simples aplicação de Y a qualquer termo λ) já que podemos trocar A por $\mathsf{f}\ \ \mathsf{A}$, ou seja, trocamos uma expressão que contém apenas A por outra que avalia f e que ainda contém A

Aplicando o combinador Y ao fatorial

UFABC

```
1  fat = \f x -> if x == 0 then 1 else x * f (x - 1)
2  Y = \f -> (\x -> f (x x)) (\x -> f (x x))
3  -- Quero calcular 3!
4  Y fat 3
5  = fat (Y fat) 3 -- Pois (Y fat) é ponto fixo de fat!
6  = (\f x -> if x == 0 then 1 else x * f (x - 1)) (Y fat)
   ↪  3
7  = (\x -> if x == 0 then 1 else x * (Y fat) (x - 1)) 3
8  = if 3 == 0 then 1 else 3 * (Y fat) (3 - 1)
9  = 3 * (Y fat 2)
10 = 3 * (fat (Y fat) 2)
11   3 * ((\f x -> if x == 0 then 1 else x * f (x - 1)) (Y
     ↪  fat) 2)
12 = 3 * ((\x -> if x == 0 then 1 else x * (Y fat) (x - 1))
   ↪  2)
13 = 3 * (if 2 == 0 then 1 else 2 * (Y fat) (2 - 1))
14 = 3 * (2 * (Y fat) 1)
15 ...
```

Aplicando o combinador Y ao fatorial

UFABC

```
1  = 3 * (2 * (Y fat) 1)
2  = 3 * 2 * ((Y fat) 1)
3  = 6 * (fat (Y fat) 1)
4  = 6 * ((\f x -> if x == 0 then 1 else x * f (x - 1)) (Y
   ↪  fat) 1)
5  = 6 * ((\x -> if x == 0 then 1 else x * (Y fat) (x - 1))
   ↪  1)
6  = 6 * (if 1 == 0 then 1 else 1 * (Y fat) (1 - 1))
7  = 6 * 1 * ((Y fat) 0)
8  = 6 * (fat (Y fat) 0)
9  = 6 * ((\f x -> if x == 0 then 1 else x * f (x - 1)) (Y
   ↪  fat) 0)
10 = 6 * ((\x -> if x == 0 then 1 else x * (Y fat) (x - 1))
   ↪  0)
11 = 6 * (if 0 == 0 then 1 else 0 * (Y fat) (0 - 1))
12 = 6 * 1
13 = 6
```

Combinador Y em Haskell

UFABC

```haskell
y = \f -> (\x -> f (x x)) (\x -> f (x x))
```

- Se você tentar criar o combinador Y em Haskell usando a expressão acima você receberá o seguinte erro:

```
• Occurs check: cannot construct the infinite type: t0 ~
  t0 -> t
 • In the first argument of ‘x’, namely ‘x’
   In the first argument of ‘f’, namely ‘(x x)’
   In the expression: f (x x)
 • Relevant bindings include
     x :: t0 -> t (bound at <interactive>:3:29)
     f :: t -> t (bound at <interactive>:3:6)
     y :: (t -> t) -> t (bound at <interactive>:3:1)
```

Combinador Y em Haskell

UFABC

- O problema ocorre pois o sistema de tipos do Haskell não permite tipos infinitos (que é justamente o que o combinador Y é!)
- Considere a expressão `(x x)`
- `x` é uma função que recebe um parâmetro
- Sejam `a` e `b` os tipos do seu parâmetro e resultado respectivamente. Logo:

```haskell
x :: a -> b
-- Como na expressão x é aplicada a x, então:
x :: a
-- Logo a = (a -> b) que é um tipo infinito
```

- Haskell não aceita tipos infinitos pois isso poderia causar problemas (laços infinitos) no verificador de tipos do compilador

Há duas soluções possíveis

- Criar um operador de ponto fixo (**veja a aula sobre cálculo λ para mais detalhes**) sem usar lambdas

```haskell
y :: (a -> a) -> a
y f = f (y f)
```

- Fazer o sistema de tipos do Haskell *engolir* os tipos na marra via `unsafeCoerce`

```haskell
import Unsafe.Coerce
y :: (a -> a) -> a
y = \f -> (\x -> f (unsafeCoerce x x)) (\x -> f
↪  (unsafeCoerce x x))
```

- Note que os tipos de `y` de ambas as soluções são exatamente os mesmos

Saída do `ghci`

```
> fat = \f x -> if x == 0 then 1 else x * f (x - 1)
> import Unsafe.Coerce
> y = \f -> (\x -> f (unsafeCoerce x x)) (\x -> f
↪  (unsafeCoerce x x))
> y fat 3
6
> y fat 10
3628800
>
```

Combinador Y em outras linguagens

UFABC

- Veja `https://rosettacode.org/wiki/Y_combinator` para uma lista da implementação do combinador Y em diferentes linguagens de programação