Paradigmas de Programação

Fabrício Olivetti de França

14 de Junho de 2018

λ-cálculo

Computabilidade

Computabilidade é uma área de estudo central da Ciência da Computação. Ela estuda a possibilidade de resolver um problema seguindo um algoritmo.

Problemas associados:

- **Decisão:** verifica se um elemento $s \in S$ está contido também em T. Exemplo: testar se um número é primo, $x \in \mathbb{N}, x \in P$.
- **Função:** calcular o resultado da aplicação de uma função $f : S \to T$. Exemplo: inverter uma *string*.
- **Busca:** verificar se xRy em uma relação binária R. Exemplo: buscar um clique em um grafo.
- **Otimização:** encontrar a solução x^* entre todas as soluções do espaço de busca S de tal forma a maximizar ou minimizar uma função $f(x)$. Exemplo: quanto devo colocar em cada possível investimento para maximizar meus lucros.

Máquina de Turing

Modelo matemático de computação criado por Alan Turing em 1936.

Consiste de uma **Máquina de Estado Finita** cuja entrada é provida por uma fita de execução de tamanho arbitrário.

A máquina permite ler, escrever, e *andar* por essa fita.

Qual a menor linguagem universal?

Nas linguagens que vocês aprenderam até então, temos:

- Atribuição (`x = x + 1`)
- Booleanos, inteiros, float, caracteres,...
- Condicionais
- Laços
- Funções
- Recursão
- Ponteiros
- Objetos, classes

Mas do que realmente precisamos para programar?

λ-cálculo

Sistema formal para expressar computação baseado em **abstração** de funções e **aplicação** usando apenas **atribuição** de nome e **substituição**.

Criado por Alonzo Church na década de 1930s.

λ-cálculo

Ele descreve computação apenas utilizando...**funções**!!

- ~~Atribuição (`x = x + 1`)~~
- ~~Booleanos, inteiros, float, caracteres,...~~
- ~~Condicionais~~
- ~~Laços~~
- **Funções**
- ~~Recursão~~
- ~~Ponteiros~~
- ~~Objetos, classes~~

Linguagem do λ-cálculo

Uma linguagem deve ser descrita em função de sua **sintaxe** e **semântica** (vocês estudarão isso em compiladores), ou como você escreve e o que significa.

Sintaxe do λ-cálculo

```
e ::= x
    | \x -> e
    | e1 e2
```

Sintaxe do λ-cálculo

Um programa é definido por uma **expressão** e, ou **termos**-λ que podem assumir uma de três formas:

- **Variável:** `x, y, z`, um nome que assumirá um valor durante a computação.
- **Abstração:** ou **função anônima** ou **função** λ, `\x -> e`, para qualquer valor `x` compute e
- **Aplicação:** `e1 e2`, aplique o argumento e2 na função e1 (`e1(e2)`). Todo `e_i` é uma expressão!

Exemplos

```
\x -> x                 -- função identidade
\x -> (\y -> y)         -- retorna a função identidade
\f -> f (\x -> x)       -- função que aplica seu argumento
                        -- (que é uma função)
                        -- na função identidade
```

Funções de dois ou mais argumentos

```
\x -> (\y -> y) -- recebe dois args e retorna o segundo
\x -> (\y -> x) -- recebe dois args e retorna o primeiro
```

Syntatic Sugar

original	syntatic sugar
`(((e1 e2) e3) e4)`	`e1 e2 e3 e4`
`\x -> (\y -> (\z -> e))`	`\x -> \y -> \z -> e`
`\x -> \y -> \z -> e`	`\x y z -> e`

```
\x y -> y
```

Escopo de uma variável

Escopo indica a visibilidade de uma variável. Em C, Java:

```
int x; /* x está visível aqui, mas y não */
{
  int y;  /* x e y estão visíveis */
}
/* y deixou de existir :( */
```

Escopo de uma variável

Na expressão `\x -> e`, *x* é uma variável e a expressão *e* é o escopo de *x*. Qualquer ocorrência de *x* em *e* está **ligada** (*bound*) por `\x`:

```
\x -> x
\x -> \y -> x
```

Escopo de uma variável

Por outro lado `x` está **livre** (*free*) se não está dentro de uma abstração:

```
x y                    -- não tem \
\y -> x y              -- x vem de outro lugar
(\x -> \y -> x) x -- o segundo x é diferente do primeiro
```

Pergunta

Na expressão `(\x -> x) x`, `x` é ligado ou livre?

Expressões fechadas

Se *e* não tem variáveis livres, então é uma **expressão fechada**.

Semântica

Podemos reescrever as expressões utilizando duas regras:

- **Passo α:** renomeia uma expressão, simplificando.
- **Passo β:** aplica uma expressão utilizando uma variável livre.

Redução β

```
(\x -> e1) e2 => e1[x := e2]
```

Toda ocorrência de `x` em `e1` é substituída por `e2`.

Redução β

```
(\x -> x) 2 => 2
```

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

Pergunta

```
(\x -> (\y -> y)) 3 => ???
```

Equivalência α

Renomeia as variáveis de uma função para evitar conflito:

```
\x -> x => \y -> y
```

Forma normal (Normal form)

Um termo λ na forma `(\x -> e1) e2` é chamado de **reducible expression** ou **redex** e pode ser reduzida utilizando um dos passos da semântica.

O termo está em sua **forma normal** se não contém nenhum **redex**.

Pergunta

Quais dos termos abaixo **não** está na forma normal?

```
x
x y
(\x -> x) y
x (\y -> y)
```

Avaliação

Um termo λ *e* é **avaliado para** *e'* se existe uma sequência de passos:

```
e => e1 => ... => en => e'
```

e *e'* é uma forma normal.

Avaliação

```
(\x -> x) 3 => 3

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

Programando com λ

Como expressamos o conceito de **Verdadeiro** e **Falso** utilizando funções?

Programando com λ

O que fazemos com **Verdadeiro** e **Falso**?

Programando com λ

O que fazemos com **Verdadeiro** e **Falso**?

Decisões no formato: `if b then e1 else e2`.

Nós já implementamos as funções necessárias para essa definição em outros slides 😄

Booleanos

```
Veradeiro = \x y -> x
Falso     = \x y -> y
IF        = \b x y -> b x y
```

Booleanos

```
IF Verdadeiro 2 3
  => (\b x y -> b x y) Verdadeiro 2 3
  => (\x y -> Verdadeiro x y) 2 3
  => (\y -> Verdadeiro 2 y) 3
  => Verdadeiro 2 3
  => (\x y -> x) 2 3
  => (\y -> 2) 3
  => 2
```

Exercício (0.5 ptos)

Definia as seguintes funções:

```
NOT = \b -> ???
AND = \b1 b2 -> ???
OR  = \b1 b2 -> ???
```

Números Naturais

Considere os números naturais $0, 1, 2, \ldots$, que operações fazemos com eles?

- Contagem: 0, `inc`, `dec`
- Aritimética: `+`, `-`, `*`
- Comparações: `==`, `<`,...

Números Naturais

Vamos começar definindo os números:

```
ZERO = ???
UM   = ???
DOIS = ???
...
```

Números Naturais

Números de Church: um número *N* é codificado como a chamada de uma função *N* vezes:

```
ZERO = ???
UM   = \f x -> f x
DOIS = \f x -> f (f x)
TRES = \f x -> f (f (f x))
...
```

Números Naturais

E o ZERO?

```
ZERO = ???
UM   = \f x -> f x
DOIS = \f x -> f (f x)
TRES = \f x -> f (f (f x))
...
```

Números Naturais

Com que essa definição parece?

```
ZERO = \f x -> x
UM   = \f x -> f x
DOIS = \f x -> f (f x)
TRES = \f x -> f (f (f x))
...
```

Números Naturais

```
ZERO  = \f x -> x
Falso = \x y -> y
```

Números Naturais

Função `INC` deve adicionar mais 1 no número `n`:

```
INC = \n -> ???
```

Números Naturais

Função `INC` deve adicionar mais 1 no número `n`:

```
INC = \n -> ???
INC ZERO = UM
```

Números Naturais

Substituindo pelas definições:

```
INC = \n -> ???
INC (\f x -> x) = \f x -> f x
```

Números Naturais

A operação que deve ser feita para encontrar o novo `x` é em função do `ZERO`:

```
INC = \n -> (\f x -> ???)
INC (\f x -> x) = \f x -> f x
INC (\f x -> x) = \f x -> f ((\f x -> x) ???)
```

Números Naturais

Se eu passar como argumento de ZERO o f e o x, obtemos:

```
INC = \n -> (\f x -> ???)
INC (\f x -> x) = \f x -> f x
INC (\f x -> x) = \f x -> f ((\f x -> x) f x)
    => \f x -> f x
```

Números Naturais

Se eu passar como argumento de ZERO o f e o x, obtemos:

```
INC = \n -> (\f x -> f (n f x))
INC (\f x -> x) = \f x -> f x
INC (\f x -> x) = \f x -> f ((\f x -> x) f x)
    => \f x -> f x
```

Exercício (0.5 pto)

Como implementar a função ADD?

```
ADD = \n m -> ???
ADD DOIS UM = TRES
ADD (\f x -> f (f x)) (\f x -> f x) = (\f x -> f (f (f x)))
```

Recursão

Como podemos fazer chamadas recursivas se as funções são anônimas (não tem nome)?

Recursão

```
SUM = n -> ???   -- 1 + 2 + ... + n
```

Recursão

Nas nossas linguagens basta fazer:

```
sum 0 = 0
sum n = n + sum (n-1)
```

E no cálculo λ?

Recursão

```
\n -> IF (ISZERO n)
        ZERO
        (ADD n (SUM (DEC n)))
```

Recursão

SUM não existe, não tem nome ainda!

```
\n -> IF (ISZERO n)
          ZERO
          (ADD n (SUM (DEC n)))
```

Recursão

Vamos criar uma função intermediaria, chamada STEP que recebe uma expressão `rec`:

```
STEP =
   \rec -> \n -> IF (ISZERO n)
                    ZERO
                    (ADD n (rec (DEC n)))
```

Nosso objetivo é passar a definição de `rec` como parâmetro de STEP.

Fixed Combinator

Queremos criar uma função `FIX` que faça:

```
FIX STEP => STEP (FIX STEP)
```

Fixed Combinator

Dessa forma teríamos:

```
SUM = FIX STEP


SUM UM
    => FIX STEP UM
    => STEP (FIX STEP) UM
    => \rec -> \n -> IF (ISZERO n) ZERO
                (ADD n (rec (DEC n))) (FIX STEP) UM
    => \n -> IF (ISZERO n) ZERO
                (ADD n ((FIX STEP) (DEC n))) UM
    => IF (ISZERO UM) ZERO
                (ADD UM ((FIX STEP) (DEC UM)))
    => ADD UM ((FIX STEP) (DEC UM))
    => ADD UM ((FIX STEP) ZERO)
```

Fixed Combinator

Dessa forma teríamos:

```
=> ADD UM (STEP (FIX STEP) ZERO)
=> ADD UM (\rec -> \n -> IF (ISZERO n) ZERO
          (ADD n (rec (DEC n))) (FIX STEP) ZERO)
=> ADD UM (\n -> IF (ISZERO n) ZERO
          (ADD n ((FIX STEP) (DEC n))) ZERO)
=> ADD UM (IF (ISZERO ZERO) ZERO
          (ADD ZERO ((FIX STEP) (DEC ZERO))))
=> ADD UM ZERO
=> UM
```

Nosso objetivo é passar a definição de `rec` como parâmetro de `STEP`.

Y-Combinator

Criado por Haskell Curry 😄

```
FIX = \stp -> (\x -> stp (x x)) (\x -> stp (x x))
```

Y-Combinator

```
FIX STEP
  => (\stp -> (\x -> stp (x x)) (\x -> stp (x x))) STEP
  => (\x -> STEP (x x)) (\x -> STEP (x x))
  => STEP ((\x -> STEP (x x) (\x -> STEP (x x))
```