Linguagens de Programação

Lambda Cálculo Não Tipado

Samuel da Silva Feitosa

Aula 21 2022/1





O que é o Cálculo Lambda?

- É um modelo computação universal.
 - Equivalente à máquina de Turing
- Ao contrário da MT, o cálculo-λ é um modelo para linguagens de programação:
 - âmbito de variáveis
 - ordem de computação
 - estruturas de dados
 - recursão
- As linguagens funcionais são concebidas como implementações computacionais do λ-cálculo (linguagem ISWIM de Peter Landin, 1964).



Sintaxe - Termos do λ-cálculo

```
x, y, z, \dots uma variável é um termo;

(\lambda x M) é um termo se x é variável e M é um termo;

(MN) é um termo se M e N são termos.
```

exemplos de termos	não são termos
У	()
$(\lambda x y)$	$x\lambda y$
$((\lambda x y)(\lambda x (\lambda x y)))$	x(y)
$(\lambda y (\lambda x (y (y x))))$	$(\lambda x (\lambda y y))$



Compreendendo o λ-cálculo

- (λx M) é a abstração de x em M.
- (M N) é a aplicação de M ao argumento N.
- Exemplos:
 - (λx x) é a função identidade: a x faz corresponder x
 - (λx (λy x)) é a função para cada x retorna uma outra função que para cada y retorna x
- Não há distinção entre dados e programas.
- Não há constantes (e.g. números).
- Tudo são λ-termos!



Sintaxe do λ-cálculo em Haskell

Definir um tipo de dado algébrico para representar a sintaxe.

```
data Expr = Var String
Lam String Expr
App Expr Expr
deriving Show
```



Substituição

• M[N/x] é o termo resultante da **substituição** de ocorrências livres de $x \in M$ por N. $(\lambda x y)[(zz)/y] \equiv (\lambda x (zz))$ $(\lambda x y)[(zz)/x] \equiv (\lambda x y)$

Nota: apenas substitui ocorrências livres de x.

```
 [x \mapsto s]x = s 
 [x \mapsto s]y = y \qquad \text{if } y \neq x 
 [x \mapsto s](\lambda y. t_1) = \lambda y. [x \mapsto s]t_1 \qquad \text{if } y \neq x \text{ and } y \notin FV(s) 
 [x \mapsto s](t_1 t_2) = [x \mapsto s]t_1 [x \mapsto s]t_2
```



Substituição em Haskell

Substituição aplicada nos três termos válidos do λ-cálculo.



Avaliação das Expressões - β-redução

- $((\lambda x M) N) \rightarrow \beta M[N/x]$ se $BV(M) \cap FV(N) = \emptyset$
- Exemplo:

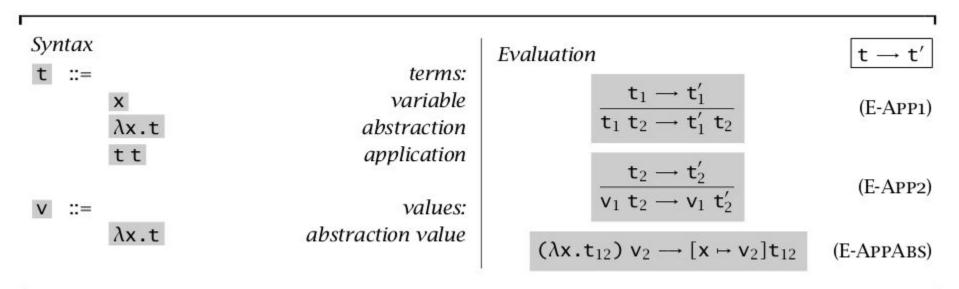
$$((\lambda x \underbrace{(x x)}_{M}) \underbrace{(y z)}_{N}) \rightarrow_{\beta} (x x)[(y z)/x] \equiv ((y z)(y z))$$

- Corresponde à invocação de uma função:
 - o x é o parâmetro formal;
 - M é o corpo da função;
 - N é o argumento.



Avaliação das Expressões - β-redução

Sintaxe e regras de avaliação do λ-cálculo.

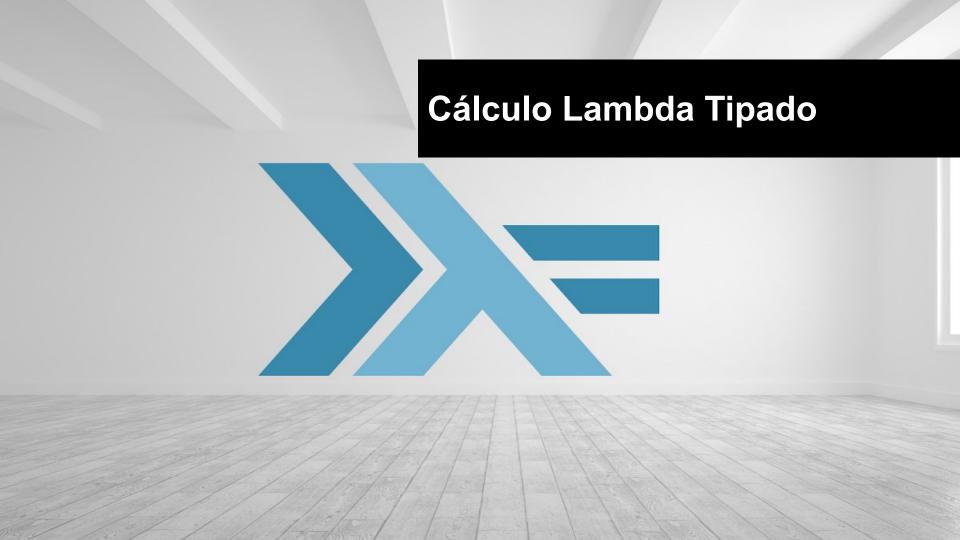




Avaliação das Expressões em Haskell

Avaliando as expressões lambda usando a estratégia call-by-value.





Tipos

- Um tipo é uma coleção de entidades computacionais que compartilham alguma propriedade comum.
- Por exemplo, o tipo int representa todas as expressões que avaliam para um inteiro, e o tipo int → int representa todas as funções de inteiros para inteiros.
- Sistemas de Tipos métodos formais leves que permitem definir/estudar o comportamento de um programa.



Usos do Sistema de Tipos

- Nomear e organizar conceitos úteis.
- Fornecer informações (para o compilador ou programador) sobre dados manipulados pelo programa.
- Assegurar que o comportamento de programas em tempo de execução seguem certos critérios.



λ-cálculo tipado

- Vamos considerar um sistema de tipos para o λ-cálculo que assegura que valores são usados corretamente.
- Por exemplo, que um programa nunca tentará usar um tipo de forma incorreta.
- A linguagem resultante deste estudo é frequentemente chamada de simply-typed lambda calculus.



λ-cálculo tipado

- No λ-cálculo tipado, o tipo do argumento é explicitamente informado pelo programador.
- Isto é, em uma abstração λx : T . e, o T é o tipo esperado no argumento da função.
- Para que isto seja possível, vamos incluir o uso de constantes booleanas no λ-cálculo não-tipado já implementado.
- A semântica operacional do λ-cálculo tipado é a mesma do não-tipado.



A relação de tipagem

- A presença dos tipos não altera a avaliação dos termos.
- Então, por quê eles são necessários?
- Usaremos os tipos para restringir quais expressões devem ser avaliadas.
 - Especificamente, o sistema de tipos do λ-cálculo tipado vai assegurar que qualquer programa bem-tipado não irá "travar".
 - Isso significa, que sempre será possível avaliar um termo bem tipado até obter um valor.
- Uma expressão e estará travada se e não é um valor e não existe uma expressão e' tal que e → e'.



Julgamento de Tipos

- Será introduzida uma relação (ou julgamento) sobre contextos de tipos (ou ambientes de tipos) Γ, expressões e, e tipos T.
- A relação Γ ⊢ e : T é lida como "e tem tipo T em um contexto Γ".
 - Um contexto de tipos é uma sequência de variáveis e seus tipos.
- Na relação de tipos F ⊢ e : T, será assegurado que se x é uma variável que aparece em e, então F associa a x um tipo.
 - O contexto de tipos pode ser visto como uma função parcial de variáveis para tipos.
- Vamos escrever **r**, **x**: **r** para indicar que o contexto de tipo é estendido com a variável **x** com tipo **r**.

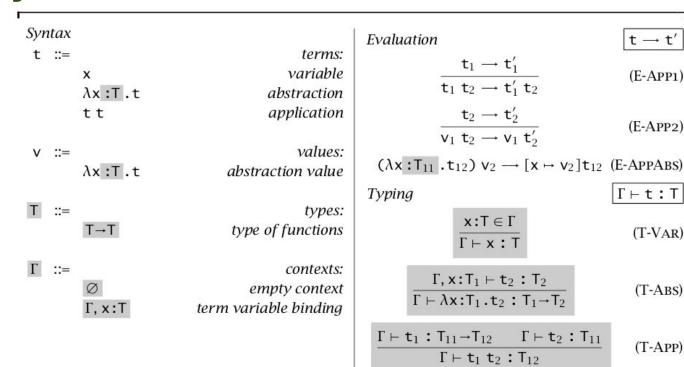


Julgamento de Tipos

- Dado um contexto de tipos Γ e uma expressão e, se existe algum T tal que Γ ⊢ e : T, dizemos que e é bem-tipado sob um contexto Γ.
- Se \(\mathbb{\Gamma}\) é o contexto vazio, ent\(\tag{a}\) o dizemos que \(\mathbb{e}\) é uma express\(\tag{a}\) o bem-tipada.



Definição indutiva de Γ ⊢ e : T



 $t \rightarrow t'$

(E-APP1)

(E-APP2)

 $\Gamma \vdash \textbf{t:T}$

(T-VAR)

(T-ABS)

(T-APP)

Verificando os tipos de uma expressão

 Vejamos uma derivação demonstrando que o termo a seguir tem o tipo Bool em um contexto de tipos vazio.

```
\frac{x:Bool \in x:Bool}{x:Bool \vdash x:Bool} \xrightarrow{T-VAR} \frac{T-TRUE}{T-ABS} = \frac{T-TRUE}{T-APP}
= \frac{(\lambda x:Bool.x) \text{ true : Bool}}{T-APP} = \frac{T-TRUE}{T-APP}
```



Implementação em Haskell

Definição dos tipos válidos:

```
data Ty = TBool
| | | | TFun Ty Ty
| deriving (Show, Eq)
```

- Note o tipo de função, que é parametrizado recursivamente.
- O contexto de variáveis pode ser definido como uma lista:

```
type Ctx = [(String, Ty)]
```



Implementação em Haskell

- Alteração na expressão que define a abstração:
 - Adição da informação do tipo.

```
data Expr = BTrue

| BFalse
| Var String
| Lam String Ty Expr
| App Expr Expr
| deriving Show
```



Implementação em Haskell

```
typeof :: Ctx -> Expr -> Maybe Ty
typeof ctx BTrue = Just TBool
typeof ctx BFalse = Just TBool
typeof ctx (Var v) = lookup v ctx
typeof ctx (Lam v t1 b) = let ctx' = (v, t1):ctx
                              Just t2 = typeof ctx' b
                            in Just (TFun t1 t2)
typeof ctx (App e1 e2) =
  case (typeof ctx e1 , typeof ctx e2) of
         (Just (TFun t11 t12) , Just t2) -> if (t11 == t2) then
                                            Just t12
                                          else
                                            Nothing
           -> Nothing
```



Considerações Finais

- Nesta aula estudamos o λ-cálculo, suas regras de avaliação e seu sistema de tipos.
 - Adaptação da AST do λ-cálculo não-tipado para definir explicitamente o tipo de um parâmetro em uma abstração.
 - Definição de um contexto de variáveis, que associa cada uma a seu tipo.
 - Definição da função eval e typeof, responsável por implementar a avaliação e a verificação estática de tipos.

