The λ-Calculus, λ-expression, and Anonymous function

王 茹 17307130285

lambda表达式， lambda演算和匿名函数

- 从编程概念上看，**lambda**表达式与匿名函数同义，都是指一个没有**标识符**的函数定义。作用是将函数作为一级值使用以简化句法。
- 从历史上看，**lambda** 表达式的起源是**Alonzo Church** 在1930s发明的**λ-calculus**。**λ-calculus** 是一个严谨的数理逻辑**形式系统**，可以被看作所有函数式语言（非函数式语言中的函数式编程特性）的理论基础，并且启发了Lisp, ML, Haskell语言的发明。Haskell语言编译器GHC就使用了扩展非静态类型特征的$System\ \ F$, 一种含类型的$\lambda - calculus$。
- lambda 演算在现代程序设计语言研究中常被用作检查新定义系统的一致性。

The untyped λ-Calculus

- 三个基本特征：函数(function)，函数应用(function application)，变量(variables)。
- 一个例子：∀**x** ∈ **A, f(x) = x**
  - **函数：(λx.x)**，其中第一个x为变量，是函数的一个参数，第二个x为函数的返回值，**.** 表示“return”。在Java中可以表示为：

    ```
    (x)->x;
    ```

  - **函数应用：f(a)** 表示为**((λx.x) a)**
  - BNF语法：

    $< expression >:=< name > | < function > | < application >$

    $< function >:= \lambda < name >. < expression >$

    $< application >:=< expression >< expression >$

- **其他例子**

  $((\lambda x.\, x)a) \to a$ by replacing x with a in x $((\lambda x.\, x)(\lambda y.\, y)) \to (\lambda y.\, y)$ by replacing x with (λy.y) in x $((\lambda y.\, (\lambda x.\, y))a) \to (\lambda x.\, a)$ by replacing y with a in (λx.y)

- **一个应用**, 返回逆序对: $\lambda(a, b).\, (b, a)$
- $M, N, L = X$

  $|(\lambda X.\, M)$

  $|(MM)$

  $X =$ a variable: x, y, ...

- **操作语义**

  - $$\begin{aligned} &\frac{}{\lambda.\ e\ \mathbf{val}} \\ &\frac{e_{lam} \mapsto e'_{lam}}{e_{lam}e_{arg} \mapsto e'_{lam}e_{arg}} \\ &\frac{}{(\lambda\, x\, e_{lam}e_{arg} \ \mapsto\ [x \to e_{arg}]e_{lam})} \end{aligned} \tag{1}$$

  x是一个元变量

- 第一条规则规定了函数式编程的铁律：只有函数是值（value）。形式为λx.e的表达式没有代入参数前无法计算。 第二条规则：化简左表达式，函数应用不变。 最后一条规则引入了一个新的构造：替换。 元语法$[x \to e_{arg}]\, e_{lam}$的意思是用$e_{arg}$代替x在$e_{lam}$中的所有实例”。
  - 操作语义的目的是将所有表达式简化为一个值（reduce expressions to values）
- **自由变量**和**非自由变量**：
  - 和所有程序设计语言一样，变量在λ-Calculus中也有作用域(scope). 形如λx. E 的函数中，如果x是新引入的变量，E就是x的作用域，此时我们称x在λx. E中被绑定了。

$$
\begin{array}{lll}
 & \text{Free} & \text{Bound} \\
(1) & \mathbf{FV}(x) = \{x\} & \mathbf{BV}(x) = \{\varnothing\} \\
(2) & \mathbf{FV}(MN) = \mathbf{FV}(M) \cup \mathbf{FV}(N) & \mathbf{BV}(MN) = \mathbf{BV}(M) \cup \mathbf{BV}(N) \\
(3) & \mathbf{FV}(\lambda x[M]) = \mathbf{FV}(M) - \{x\} & \mathbf{BV}(\lambda x[M]) = \mathbf{BV}(M) \cup \{x\}
\end{array}
$$

  - 一个lambda演算在所有变量都是绑定的时候才是合法的。因为一个自由变量并不是函数，也不是可以被化简为函数的表达式。但是内层演算允许出现自由变量。
  - 如果$FV(M) = \emptyset$ 那么 M 被称为是一个 组合子（**combinator**）。
    - $$\begin{array}{rcl} I &=& \lambda x.\, x \\ S &=& \lambda f.\, \lambda g.\, \lambda x.\, fx(gx) \\ K &=& \lambda x.\, \lambda y.\, x \\ Y &=& \lambda f.\, (\lambda x.\, f(xx))(\lambda x.\, f(xx)) \end{array}$$
    - 定理：所有组合子都可以由S，K构成。（BCKI）
      - I = SKK
  - λx. x (**λx. x**) x (**variable shadowing**)：
    - 内层函数的同名变量可能与外层含义不同
- **化简规则**：

$$
\begin{array}{rcl}
(\lambda X1.M) & \alpha & (\lambda X2.M[X1 \leftarrow X2])\ where\ X2 \notin FV(M) \\
((\lambda X.\, M1)M2) & \beta & M1[X \leftarrow M2] \\
(\lambda X.\, (MX)) & \eta & M\ where\ X \notin FV(M)
\end{array}
$$

- α-equivalence可以解释为换命规则，即替换绑定变量的名字并不修改函数的语义。
- β-equivalence可以解释为函数的应用。
  - Church-Rosser定理：If $M =_n N$, then there exists an L' such that $M \to\to_n L'$ and $N \to\to_n L'$
- 如果不能通过以上规则化简，我们称该表达式为一个标准型（normal form）
  - If e is in normal form and $e \to_{\beta *} f$ then e is identical to f.
  - 我们不知道一个表达式是不是有β normal form
    - halting problem
    - means to prove a single expression
- λ-Calculus的计算能力与图灵机等效（Turing complete）。也就是说，使用λ-Calculus的定义和语法可以模拟所有的计算机程序，反之亦然。理论上来讲，函数式编程语言与面向对象或者过程式语言的能力是一样的。为了展示λ-Calculus的实际能力，我们来看几个模拟计算的例子。
  - 定义布尔变量
    - TRUE = $\lambda x.\, \lambda y.\, x$ = K
      - **const** in haskell
    - FALSE = $\lambda x.\, \lambda y.\, y$ = KI
  - 定义操作符（使用组合子）

        - NOT = $\lambda b.b$ FALSE TRUE
        - If = □ □$\lambda l.\lambda n.lmn$
    - 定义自然数：
        - $0 =_{def} \lambda f.\lambda s.s$
        - $1 =_{def} \lambda f.\lambda s.fs$
        - $2 =_{def} \lambda f.\lambda s.f(fs)$
- 用lambda演算模拟递归：
    - a fixed point combinator **Y**: $Y. = \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$, 实际上是Haskell Curry发明的。
    - $Yf \equiv f(Yf)$
        - ```
          const Y = f => (x => x(x))(x => f(y => x(x)(y)));
          const factorial = f => (x => (x === 1?1:x*f(x-1)));
          const Yfactorial = Y(factorial)(10)
          ```

The typed λ-Calculus

-

$Reference$

https://www.cs.utah.edu/~mflatt/past-courses/cs7520/public_html/s06/notes.pdf

https://en.wikipedia.org/wiki/Lambda_calculus

https://cs242.stanford.edu/f19/lectures/02-1-lambda-calculu
          ```