Universidad Nacional Autónoma de México
Facultad de Ciencias

Lenguajes de Programación

Karla Ramírez Pulido

Cálculo Lambda

Introducción

- La función $x \rightarrow x^2$ se denota como

```
(lambda (x) (*  x  x))
```

- La función $y \rightarrow (3 + y) * (y - 4)$ es

```
(lambda (y)  (*  (+  3  y) (-  y  4)))
```

- La función $(x, y) \rightarrow x * y$ se define como

```
(lambda (x  y) (*  x  y))
```

Cálculo Lambda

Variables

v

Abstracciones Lambda

(lambda (x) x)

Aplicaciones de función

((lambda (x) x) 7)

Contexto de LISP

Al respecto el mismo McCarthy dice *“Para usar funciones como argumentos, uno necesita una notación para funciones, y parecía natural usar la notación λ de Church (1941). Yo no entendí el resto de su libro de manera que no intenté implementar su mecanismo general para definir funciones”. Esto muestra la humildad de un gran científico, LISP tuvo muchas innovaciones tanto teóricas como prácticas.*

LISP (McCarthy)

- El uso de recolección de basura como un método para reutilizar celdas de memoria.
- El uso de cerraduras para implementar alcance estático.
- La invención de la expresión condicional y su uso para escribir funciones recursivas.
- El uso de funciones de orden superior, es decir, funciones que reciben funciones como argumento y/o devuelven funciones como resultado.

Cálculo Lambda Puro λ^U

El cálculo lambda es un formalismo matemático inventado durante finales de la década de l930 por Alonzo Church, Stephen Kleene y Haskell Curry (lógica combinatoria) con el propósito de fundamentar la matemática.

Si bien tal propósito no se cumplió, el mismo Church dijo alguna vez que el sistema podría tener otras aplicaciones distintas a su uso como una lógica. Palabras proféticas dado que el cálculo lambda, diseñado para investigar la definición y aplicación de funciones así como diversos principios de recursión, se ha convertido en el cálculo núcleo para estudiar fundamentos de **lenguajes de programación.**

Cálculo Lambda Puro λ^U

e ::= x

| λx.e

| e e

- (lambda (x) e)
- (e1 e2)

La aplicación se asocia a la izquierda de manera que:

e1 e2 e3 significa (e1 e2) e3

y NO e1 (e2 e3)

Abstracción Lambda

λx.e denota la función $x \rightarrow e$

que asocia a cada valor x la expresión e.

Para facilitar la abstracción lambda:

$$\lambda\ x1\ x2\ \ldots\ xn.e\ =_{def}\ \lambda x1.\lambda x2.\ \ldots\ .\lambda xn.e$$

Abstracción Lambda

$$\lambda\ x1\ x2\ \ldots\ xn.e \ =_{def}\ \lambda x1.\lambda x2.\ldots.\lambda xn.e$$

1. $\lambda xyz.xz(yz) = \lambda x.\lambda y.\lambda z.xz(yz)$

   $= \lambda x.\lambda y.\lambda z.(xz)(yz)$

1. $(\lambda xyz.xz)yz = ((\lambda xyz.xz)y)z$

   $= ((\lambda x.\lambda y.\lambda z.xz)y)z$

Noción de variables libres y ligadas

En la abstracción **λx.e** el operador λ liga a la variable **x** en e se mantiene la noción de variables libres y ligadas introducida anteriormente con las expresiones **let**.

El conjunto de variables libres de una abstracción se define como

$$\mathcal{FV}(\lambda x.e) = \mathcal{FV}(e) \backslash \{x\}.$$

$\mathcal{FV}$= Free Variable

Combinador

Una expresión e tal que si $\mathcal{FV}(e) = \varnothing$ es una expresión cerrada también conocida como **combinador**.

Algunos términos del cálculo λ y su significado intuitivo son:

- x, la función indeterminada x
- xy, la aplicación de la función x a la función y, ambas indeterminadas.

- xyz, la aplicación de la función xy a z, todas indeterminadas.
- x(yz), la aplicación de x a la aplicación de y a z.
- λx.x la función que toma un argumento x y devuelve el mismo elemento x, es decir, la función identidad.
- λz.y, la función que toma un argumento z y devuelve y, es decir, la función constante y.
- λx.λz.x, la función que para cada x, devuelve la función que para cada z devuelve x, es decir, la función que para cada x devuelve la función constante que devuelve x.

- λx.λy.λz.xyz, la función que le aplica a cada z el resultado de aplicarle x a y.
- λx.λy.λz.x(yz), la función composición de x y y.
- λx.xx, la función que para cada x devuelve la aplicación de x a x.

Semántica Operacional

$(\lambda x.t)\ s \quad \rightarrow_{\beta} \quad t\ [x := s]$

Redex: $(\lambda x.t)$

Reducto: $t\ [x := s]$

➔ $x\ [x := r] = r$

➔ $y\ [x := r] = y \quad \text{si } x \neq y$

➔ $(ts)[x := r] = t\ [x := r]\ s\ [x := r]$

➔ $(\lambda y.t)[x := r] \ = \ \lambda y.t[x := r]$

donde s.p.g. $y \neq x$ y $y \notin (r)$

Relación de Reducción o Evaluación del Cálculo Lambda $\rightarrow_{\beta}$

$e \rightarrow_{\beta} e'$ syss en e existe un redex $r =_{def} (\lambda x.t)s$ y e' se obtuvo a partir de e sustituyendo r en e por su reducto r'

$$r' =_{def} t\,[x := s]$$

Ejemplos:

1. $(\lambda x.x\ (xy))\ r \rightarrow_\beta r\ (ry)$
2. $(\lambda x.y)\ r \rightarrow_\beta y$
3. $(\lambda x.(\lambda y.yx)\ z)\ v \rightarrow_\beta (\lambda y.yv)\ z$
4. $(\lambda y.y\ v)\ z \rightarrow_\beta z\ v$

Formas Normales

Dada e si no existe e' tal que e $\rightarrow$ e' entonces decimos que e está en Forma Normal.

Si e $\rightarrow^*_\beta$ e_f y e_f está en Forma Normal entonces decimos que e_f es la Forma Normal de e.

¿Cuál es la FN de las siguientes expresiones?

1. x
2. xy
3. λx.xz

Todas están en F. N.

¿Cuál es la FN de las siguientes expresiones?

4.$(\lambda x.y)(\lambda x.x) \rightarrow_\beta y\,[x:=(\lambda x.x)] \rightarrow_\beta y$

5.$(\lambda x.(\lambda y.yx)z)v \rightarrow_\beta (\lambda y.yv)z$

es decir ahora volvemos a aplicar la beta reducción de la siguiente forma:

$\rightarrow_\beta (\lambda y.yv)z \rightarrow_\beta zv$

No terminación del Cálculo Lambda Puro

La propiedad de terminación no es válida en el Cálculo Lambda puro, al existir expresiones e que no cuentan con una F.N.

$$\text{Sean} \quad \omega =_{def} \lambda x.xx \quad y \quad \Omega =_{def} \omega\omega$$

La semántica operacional nos lleva a la siguiente secuencia de reducción:

$$\Omega =_{def} \omega\omega = (\lambda x.xx)\omega \rightarrow_{\beta} (xx)[x := \omega] = \omega\omega =_{def} \Omega$$

Combinador Ω

De manera que Ω se reduce así mismo en un paso lo cual genera la sucesión infinita de reducción:

$$\Omega \rightarrow_{\beta} \Omega \rightarrow_{\beta} \Omega \rightarrow_{\beta} \Omega \rightarrow_{\beta} \ldots$$

Ω no tiene F.N.

Confluencia

El Cálculo Lambda tiene una naturaleza no determinista, por ejemplo la expresión $(\lambda x.(\lambda y.yx)z)v$

tiene dos reductos distintos:

$(\lambda x.(\lambda y.yx)z)v \rightarrow_\beta (\lambda y.yv)z$ al reducir el redex $(\lambda x.(\lambda y.yx)z)\ v$

$\rightarrow_\beta zv$

$(\lambda x.(\lambda y.yx)z)v \rightarrow_\beta (\lambda x.zx)v$ al reducir el redex $(\lambda y.yx)\ z$

$\rightarrow_\beta zv$

Teorema 1 (Confluencia o propiedad de Church-Rosser)

Si $e \rightarrow^*_\beta r$ y $e \rightarrow^*_\beta s$ entonces

existe un término t tal que $r \rightarrow^*_\beta t$ y $s \rightarrow^*_\beta t$

Corolario 1 (Unicidad de las formas normales)

Si e tiene una Forma Normal entonces es única.

Es decir si $e \rightarrow^*_\beta e_f$ y $e \rightarrow^*_\beta e'_f$

entonces $e_f = e'_f$ (salvo α-equivalencia).

Alpha conversiones

La regla de alfa-conversión fue pensada para expresar la idea siguiente: los nombres de las variables ligadas no son importantes.

Por ejemplo, si reemplazamos *x* por *y* en $\lambda x.\lambda y.x$

obtenemos $\lambda y.\lambda y.y$

que claramente, **no es la misma función**. Este fenómeno se conoce como **captura de variables**.

Reemplazando x por z y manteniendo el mismo significado:

$$\lambda x.\lambda y.x \Rightarrow \lambda z.\lambda y.z$$

La regla de alfa-conversión establece que si *V* y *W* son variables, *E* es una expresión lambda, y *E*[*V*:= *W*]

representa la expresión *E* con todas las ocurrencias libres de *V* en *E* reemplazadas con *W*, entonces

$$\lambda V.E == \lambda W.E[V := W]$$

si *W* no está libre en *E* y *W* no está ligada a un λ donde se haya reemplazado a V. Esta regla nos dice, por ejemplo, que λ *x*. (λ *x*. *x*) *x* es equivalente a λ *y*. (λ *x*. *x*) *y*.

Numerales de Church

$0 =_{def} \lambda s.\lambda z.z$

$1 =_{def} \lambda s.\lambda z.sz$

$2 =_{def} \lambda s.\lambda z.s(sz)$

$3 =_{def} \lambda s.\lambda z.s(s(sz))$

$n =_{def} \lambda s.\lambda z.\ s\ (\ldots\ (s\ z)\ \ldots)$

Representación de booleanos, números naturales y pares

Booleanos

$\text{true} =_{\text{def}} \lambda x.\lambda y.x$

$\text{false} =_{\text{def}} \lambda x.\lambda y.y$

$\text{ift} =_{\text{def}} \lambda v.\lambda t.\lambda f.vtf$

Azúcar Sintáctica

$$\text{fun}(x) \Rightarrow e \quad =_{\text{def}} \quad \lambda x.e$$

- fun(x) ⇒ 7 es la función constante 7 i.e. λx.7
- fun(x) ⇒ (fun(y) ⇒ x) es decir, λx.λy.x
- fun (x) ⇒ if x then false else true es la función negación ¬x, λx.if x then false else true

Otro ejemplo:

La composición de funciones λf.λg.λx.f(gx)

se puede declarar como

fun (f) ⇒ (fun (g) ⇒ (fun (x) ⇒ f (gx)))

Más azúcar sintáctica

fun (x1, … ,xn) ⇒ e $=_{def}$

fun (x1) ⇒ (fun (x2)⇒ … (fun (xn) ⇒ e) ...) $=_{def}$ λx1.λx2....λxn.e

De esta manera la composición de funciones queda definida como:

fun(f, g, x) ⇒ f(g x)

Azúcar sintáctica: let

let x = e_1 in e_2 end

$=_{def}$ $(\lambda x.e_2)\ e_1$

Ejemplo:

let x = 2

in x + x

end

$=_{def}$ $(\lambda x.x+x)\ 2$

La importancia de nombrar funciones

fun factorial (n) ⇒ e

factorial (0) = 1

factorial (n) > 0

n * factorial (n - 1)

En general para declarar funciones: fun f(x) ⇒ e

Declaración de funciones usando let

let fun f(x) ⇒ e_1 in e_2 end $=_{def}$

let f = (fun (x) ⇒ e_1) in e_2 end

Ejemplo 1:

let fun **doble** (x) ⇒ x + x in doble 2 end

let fun f(x) ⇒ e_1 in e_2 end $=_{def}$

let f = (fun (x) ⇒ e_1) in e_2 end

let fun **doble** (x) ⇒ x + x in doble 2 end $=_{def}$

let **doble = (fun (x) ⇒ x + x) in double 2 end**

Recordemos que let es azúcar para (λx.e2)e1

$=_{def}$ (λ doble.doble 2)(λx.x+x)

(λdoble.doble 2)(λx.x+x) → (doble 2)[doble:= (λx.x+x)]

→ (λx.x+x)2 → 2 + 2 → 4

Ejemplo2:

let fun fac (x) $\Rightarrow$ if iszero x then 1 else x $*$ fac (x−1)
in fac 2 end

$=_{def}$

$(\lambda fac.fac\ 2)(\lambda x.$ if iszero x then 1 else x $*$ fac (x−1))

$\rightarrow$

(fac 2) [fac := $(\lambda x.$ if iszero x then 1 else x $*$ fac (x−1))]

$(\lambda x.\ \text{if}\ \text{iszero}\ x\ \text{then}\ 1\ \text{else}\ x * \text{fac}\ (x-1))\ 2$

$\rightarrow$ if iszero 2 then 1 else 2 $*$ fac (2−1)

$\rightarrow$ if false then 1 else 2 $*$ fac (2−1)

$\rightarrow$ 2 $*$ fac (2−1) +

Ya no podemos seguir sustituyendo porque no tenemos el valor de fac en fac(2-1)

Un poco de historia

Mitchell cuenta, que las funciones recursivas eran nuevas cuando LISP apareció. McCarthy, además de incluirlas en LISP, fue el principal promotor para agregar funciones recursivas a Algol60. Fortran, en comparación, no permitía que una función se llamara así misma.

Los miembros del comité para Algol60 escribieron más tarde que no tenían idea de por qué aceptaron incluir funciones recursivas en Algol60.

McCarthy argumenta que la notación lambda es inadecuada para expresar funciones recursivas. Esta afirmación es falsa. El Cálculo Lambda puro y por lo tanto LISP, es capaz de expresar funciones recursivas sin utilizar ningún operador adicional.

Recordemos que el Cálculo Lambda puro permite la autoaplicación con la que podemos definir combinadores de punto fijo.

Recordemos que x es punto fijo de una función F si y sólo si

$$F(x) = x$$

Definición 1 PUNTO FIJO

Un λ-término cerrado F es un combinador de punto fijo syss cumple alguna de las siguientes condiciones:

1. $F\ g \rightarrow_{\beta} g\ (Fg)$
2. $F\ g =_{\beta} g\ (Fg)$ es decir, existe un término L tal que

$$F\ g \rightarrow_{\beta} L \qquad y \qquad g\ (Fg) \rightarrow_{\beta} L$$

Ejemplo: función factorial

$$fac = F\ g \qquad \text{donde}$$

$$g = \lambda f \lambda n.\text{if } n = 0 \text{ then } 1 \text{ else } n * f\ (n-1)$$

Lenguaje ISWIM

En los años 50 y 60 se descubrió la conección entre los lenguajes de programación y varios aspectos del Cálculo Lambda, esto debido al deseo particular de especificar el significado del lenguaje Algol60 y de formalizar el estudio de los lenguajes de programación empleando sistemas matemáticos conocidos.

Lenguaje ISWIM

Uno de los primeros lenguajes de programación formales, basado en el Cálculo Lambda, es ISWIM, creado por Peter Landin. Si bien este lenguaje, o más bien esta familia de lenguajes, nunca fue implementada directamente, sirvió como base para explorar aplicaciones e implementaciones mediante máquinas abstractas o virtuales. El lenguaje real más cercano a ISWIM es Scheme.

Un acrónimo para la frase: “*If you See What I Mean*”.

Definición de ISWIM

e ::= x

| λx.e

| e e

| c

| $o(e_1,...,e_n)$

c: es una constante primitiva

o: es un operador primitivo,

El primero tomado de un conjunto de constantes C y el segundo un conjunto de operadores dados O.

Por ejemplo, la instancia de ISWIM relacionada a nuestro lenguaje CAE de expresiones aritméticas y booleanas consiste:

C = {true, false, 0, 1, 2...}

O = {suc,pred,iszero,if,suma,prod}

Semántica operacional de ISWIM

La semántica operacional→se define como la unión de la β-reducción con la llamada δ-reducción, es decir

$\rightarrow =_{def} \rightarrow_{\beta} \cup \rightarrow_{\delta}$ donde la δ-reducción define el comportamiento de los operadores primitivos. Ejemplos:

$$suma(m, n) \rightarrow_{\delta} m + n$$

$$if\ (false, e2, e3) \rightarrow_{\delta} e3$$

Reglas de evaluación

Si $o^{(n)} \in O$ entonces agregamos las siguientes reglas δ:

$$\frac{e_1 \rightarrow_\delta e'_1}{o(e_1,...,e_n) \rightarrow_\delta o(e'_1,...,e_n)}$$

$$\frac{e_2 \rightarrow_\delta e'_2}{o(v_1,...,e_n) \rightarrow_\delta o(v_1,...,e_n)}$$

$$\frac{e_n \rightarrow_\delta e'_n}{o(v_1,...,v_{n-1},e_n) \rightarrow_\delta o(v_1,...,v_{n-1},e'_n)}$$

Ejemplo:

- suma (suma(1, 2), suma(2, 2))

recordemos que la expresión es de la forma: $o(e_1, e_2, \ldots, e_n)$

- Donde o = suma, e_1 = suma(1, 2) y e_2 = suma(2, 2)

suma (1,2) $\rightarrow_{\delta}$ 1 + 2 = 3

---

suma (suma(1,2), suma(2,2)) $\rightarrow_{\delta}$ suma (3, suma(2,2))

- suma (suma(1, 2), suma(2, 2))

suma (2,2) $\rightarrow_{\delta}$ 2 + 2 = 4

---

suma (3, suma(2,2)) $\rightarrow_{\delta}$ suma(3, 4)

ahora suma(valor1, valor2) entonces suma(3, 4) ⇒

suma(3, 4) $\rightarrow_{\delta}$ 3 + 4 = 7

---

suma(3, 4) $\rightarrow_{\delta}$ suma (3, 4)

Gracias