Programación Funcional

-- Cálculo Lambda (λc) --

El λc fué introducido por Alonso Church en la década de 1930 para establecer una teoría formal de *funciones computables* que permitió demostrar por primera vez un teorema de indecibilidad. Este cálculo que consiste en una sola regla de transformación basada en sustitución de variables y un esquema de definición de función única, es considerado como un lenguaje de programación universal.

Este lambda cálculo (λc) es universal en el sentido a que cualquier *función computable* puede expresarse y evaluarse usando este formalismo, haciéndolo equivalente a la Máquina de Turing presentada en 1936. Sin embargo, el λc enfatiza el uso de *reglas de transformación* sin hacer referencia a la máquina real que las implementa, siendo así un enfoque más relacionado software que con hardware (Michaelson, 2011).

Luego, el objetivo del λc es emplear funciones como medio de transformación de argumentos en resultados. A este formalismo se denomina *cálculo* ya que emplea un conjunto de axiomas y reglas de inferencia para representar el medio de transformación mencionado.

Contenido

Sintaxis del λc

La sintaxis abstracta de λc *puro sin tipos* incorpora lo que se llama *expresión lambda* (λe) M que se define por la siguiente gramática recursiva:

M → x | λx.M | M_1M_2 donde

- x es una variable
- λ se usa para marcar el nombre del *parámetro* de la función.
- La expresión **λx.M** es una ***abstracción funcional***. Define a una función con parámetro

x y cuerpo M, donde M especifica cómo se organizan los argumentos.

- M_1M_2 es una ***aplicación funcional***. Define la aplicación de la función M_1 a M_2.

Adicionalmente, para completar la definición del λc se definen tres reglas simples de reescritura en expresiones lambda: α-conversión, **β-reducción**, η-conversión.

Estas reglas, junto con las reglas de relación de equivalencia estándar para reflexividad, simetría y transitividad, inducen una teoría de la convertibilidad en el cálculo lambda, que es consistente como un sistema matemático.

Notación

Como características de λc se tiene que:

- Las funciones son anónimas
- El nombre de las variables es irrelevante.
- Se basa en funciones con un solo valor de entrada. Luego, una función puede aceptar como argumento a otra función solo si esta última tiene un solo argumento.

Luego, cuando se requiere trabajar con múltiples argumentos, **se debe utilizar una técnica** llamada *currying* o *currificación* la cual transforma los argumentos de entrada a una función en una cadena de funciones de un solo valor. Por ejemplo, una función

[f(x, y) → x + y]	puede ser reescrita de la siguiente forma: [x → (y → x + y)] donde el operador → asume una asociación a la derecha. Si x = 2 se obtiene la función y → 2 + y. Si y = 3 se obtiene 2+3=5.

En λc las funciones están definidas por λe (lambda expresiones) que indican el comportamiento de su argumento. Por ejemplo, la función

f(x) = (x+2)	En λc se expresa como la abstracción **λx.(x+2)** donde **x** es el parámetro de la función y **(x+2)** el cuerpo M.

La instanciación	Se usa la aplicación funcional
f(3) = 3+2	En λc: Se expresa a 3 como una función constante N.
	Entonces (λx.(x+2))3 = MN = 5.
	Aplicación de: (λx.M)N → [N/x]M

Características de las variables

En el λc como en la mayoría de los lenguajes de programación, todas las variables son locales a sus definiciones y en este contexto son de dos tipos: ligadas y libres, dependiendo de que sean o no definidas como parámetros de una función (Que sean definidas con el símbolo ʎ en la abstracción funcional).

Por ejemplo, en la función ***λx****.M* se dice que *x* está *ligada* ya que su aparición en el cuerpo de la definición está precedida por ***λx***. Por el contrario, una variable no precedido por *λ* es llamada variable *libre*. Por su parte, en la expresión *λx.xy* la variable *x* está ligada y la variable *y* es libre.

En la expresión *(λx.x)(**λ**y.yx)* la *x* en el cuerpo de la primera expresión de la izquierda está ligada a la primera λ. La *y* en el cuerpo de la segunda expresión está ligada a la segunda *λ* y *x* es libre. Es de anotar que la *x* en la segunda expresión es totalmente independiente de la *x* en la primera expresión.

También se puede dar el caso en que el mismo identificador puede estar libre y ligado en la misma expresión (λx**.**xy)(λy.y). Como se observa, la primera *y* es libre en la sub-expresión entre paréntesis de la izquierda, y se encuentra ligada en la sub-expresión de la derecha.

A continuación se plantean reglas para variables libres:

- En una expresión x, la variable x es libre. Es decir libre(x) = { x }
- En la expresión *λx.M* cada *x* en *M* está vinculada. Cada variable que no sea *x* es libre en M y en *λx.M*. Cada variable que esté vinculada en M también lo está en λx.M.

- Todas las variables libres de MN son el resultado de la unión de dos conjuntos: las variables libres de M y las variables libres de N. Es decir, libre(MN) = libre(M) U libre(N).

- Las variables vinculadas de MN son también el resultado de la unión de dos conjuntos: las variables vinculadas de M y las variables vinculadas de N.

Reglas de reescritura en λc

En el λc se definen tres *reglas de reescritura* basadas en sustitución de variables, lo que permite implementar la transformación de argumentos en resultados.

α-conversión

Esta regla plantea que dos términos *s* y *t* se dice que son α-convertibles si uno es obtenido del otro mediante el renombramiento de variables ligadas.

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

Luego x = z. Como se observa, la regla de α-conversión permite concluir que los nombres de las variables ligadas o parámetros formales no son importantes.

β-reducción (computar o calcular o instanciar)

Es una regla de transformación de reescritura mediante sustitución de variables, que expresa la idea de la aplicación funcional de la siguiente manera:

$(\lambda x.M)N \rightarrow [N/x]M$ la cual quiere decir que $(\lambda x.M)N$ puede ser reducida a $[N/x]M$,

donde [N/y] significa: "el término que se obtiene al sustituir N en el lugar de y, toda vez que y *de N* ocurre libre en M".

Ejemplo1.

Sea $(\lambda x.x+1)$ Abstracción de función

Sea $(\lambda x.x+1)3$ Aplicación de la abstracción a un parámetro N.
N es la función 3. Luego es de la forma MN:

En este caso las variables libres de N (que no tiene porque es constante) no tienen apariciones ligadas en M

Evaluación: $[3/x]x+1$
$= 3 +1 = 4$

Ejemplo2.

Sea (λx.x+1)**(**(λy.y+2) 3**)** (λx.x+1)(MN)

MN = [N/y]M = [3/y](y+2) = 3+2=5

(λx.x+1)5 = [N/x]M = [5/x](x+1) = 5+1= 6

Ejemplo3.

Sea **(**λf.λx.fx**)**λy.y+1 Es de la forma (λf.M)N donde M**=** λx.fx

N = λy.y+1

[(λy.y+1)/f] λx.**fx**

= λx [(λy.y+1)/ f] fx

= λx.(λy.y+1)x que es de la forma λM.N

Luego **=** [x/x] λy.y+1

= λy.y+1

Reglas de la β-reducción

M → x | λx.M | M_1M_2

Las posibles sustituciones para **(λx.M) N → [N/x]M** dependen de la ocurrencia de las variables libres de N en M, y se plantean mediante las siguientes reglas:

Sea [N/x]M

(i) Cuando M sea la variable a sustituir: **[N/x]M** : [N/x] x = **N**

Ejemplo4 Reducir x λx.mn

MN= x λx.mn → [λx.(mn)/x] x = λx.(mn) [(mn)/t] t= mn

M → x | λx.M | M_1M_2

(ii) Cuando M sea una variable, pero diferente de la sustituida: [N/x] y = y Si y ≠ x

Ejemplo5 Reducir **s** λy.(mn)

MN= s λy.(mn) → [λy.(mn)/t] s = **s**

(iii) Cuando M sea una aplicación: [N/x](PQ) = ([N/x]P [N/x]Q)

Ejemplo6 Reducir yr λy.(mn) **M → x | λx.M | M_1M_2**

[λy.(mn)/y**]**(yr)

= ([λy.(mn) / y] y [λy.(mn) / y] r)

??

donde **en general** [x/m] (λy.(mn) m) = [x/m] λy.(mn) [x/m] m

(iv) Cuando M sea una abstracción: [N/x] (λx.P) = λx.P "si x no es libre"

Ejemplo7 Reducir λx.(tx) λy.(mn)

[λy.(mn)/x] λx.(tx) = λx.(tx)

[x/m] λm.(mn)= λm.(mn)

(v) Cuando M sea una abstracción: [N/x] λy.M = λy.[N/x] M "Si x ≠ y ∧ y ¬Є libre(N)"

Ejemplo8 [λn.(mn)/x] λy.(xy) = λy.[λn.(mn)/x] (xy) y no ocurre libre en N

(vi) Cuando M sea una abstracción: [N/x](λy.M) = λz.[N/x] [z/y] M

Si y ≠ x ∧ z ¬Є libre(N) ∧ z ¬Є libre(M)

Ejemplo9 [λn.(my)/x] λy.(xy) = λz.[λn.(my)/x] (xy)

En el enfoque funcional, la evaluación de una expresión se entiende como la aplicación de una función a determinados argumentos efectivos, lo que en el λc se denomina *redex*. Un redex es un término de la forma (λx.MN). Una forma normal de una expresión lambda es un término que no contiene ningún *redex*, es decir una expresión que no admite ninguna reducción.

Como estrategias de reducción tenemos las evaluaciones de orden normales y aplicativas. La estrategia de evaluación de orden normal consiste en reducir siempre el *redex* de más a la izquierda, mientras que la de orden aplicativo consiste en reducir primero los dos términos del redex antes que la aplicación que él denota sea reducida.

η-conversión (extensionalidad).

La eta conversión expresa que dos funciones son la misma ssi dan el mismo resultado para cualquier argumento.

λx.(Ex) E Si x Є fv(E)

La consistencia matemática del λc requiere que la *aplicación funcional* y el resultado puedan ser interpretadas como expresiones equivalentes: (MN)=R, es decir representan el mismo valor. En este contexto, la *relación de equivalencia* entre expresiones del λc tendrá las siguientes propiedades:

- Reflexividad: M = M

- Simetría: M = N → N = M

- Transitividad: M = N ∧ N = P → M = P

- Mas las siguientes que resultan lo suficientemente intuitivas:

M = N → (MP) = (NP) M = N → (PM) = (PN) M = N → λx.N = λx.M

Cálculo lambda en Python: Funciones anónimas

En Python podemos crear una función, usarla solo una vez y darle un nombre por esta vez puede ampliar innecesariamente el espacio de nombres del programa.

Es posible entonces definir una función anónima. En Python, una función anónima es aquella que no tiene nombre y solo se puede usar en el punto en que se define. Las funciones anónimas se definen usando la palabra clave *lambda* y por este motivo también se conocen como funciones lambda.

La sintaxis utilizada para definir una función anónima es:

lambda argumentos: expresión

Las funciones anónimas pueden tener cualquier número de argumentos, pero solo una expresión (que es una declaración que devuelve un valor) como su cuerpo. La expresión se ejecuta y el valor generado a partir de ella se devuelve como resultado de la función.

Como ejemplo, definamos una función anónima que cuadrará un número:

```
double = lambda i: i*i
```

En este ejemplo, la definición lambda indica que hay un parámetro para la función anónima *('i')*, que el cuerpo de la función se define después de los dos puntos ':' que multiplica i * i cuyo valor se devuelve como resultado de la función.

Toda la función anónima se almacena en una variable llamada *double*. Podemos almacenar la función anónima en la variable, ya que todas las funciones son instancias de la función de clase y pueden ser referenciadas de esta manera.

Para invocar la función, podemos acceder a la referencia a la función contenida en la variable double y luego usar los corchetes para hacer que la función se ejecute, pasando los valores que se utilizarán para los parámetros:

```
print(double(10))
```

Cuando esto se ejecuta, se imprime el valor 100.

A continuación se proporcionan otros ejemplos de funciones lambda / anónimas (que ilustran que una función anónima puede tomar cualquier número de argumentos):

```
func0 = lambda: print('no args')
func1 = lambda x: x * x
func2 = lambda x, y: x * y
func3 = lambda x, y, z: x + y + z
```

Estos se pueden usar como se muestra a continuación:

```
func0()
print(func1(4))
print(func2(3, 4))
print(func3(2, 3, 4))
```

Otros ejemplos:

```
fun = lambda a : 1+a
```

```
print(fun(3)) # Resultado: 4


fun = lambda a, b : a+b
print(fun(3, 4)) # Resultado: 7
```

Referencias

[1] Gómez Perdomo, J., Castro Rojas, W., & Cardona López, A. Programacion funcional y lambda cálculo. *Ingeniería e Investigación; núm. 40 (1998); 72-82 Ingeniería e Investigación; núm. 40 (1998); 72-82 2248-8723 0120-5609*.

[2] Hunt, J. (2019). *Advanced Guide to Python 3 Programming*. Springer International Publishing.

[3] Michaelson, G. (2011). *An introduction to functional programming through lambda calculus*. Courier Corporation.

[4] Osorio, F. A. G. (1998). Programación funcional: conceptos y perspectivas. *Ingeniería e Investigación*, (40), 65-71.

[5] Rojas, R. (2015). A tutorial introduction to the lambda calculus. *arXiv preprintarXiv:1503.09060*.

---------- Fin de documento