UNCUYO
UNIVERSIDAD
NACIONAL DE CUYO

Paradigmas de Programación
Paradigma Funcional

Dr. Pablo Vidal

Facultad de Ingenieria
Universidad Nacional de Cuyo

2022

Tabla de Contenidos

Cálculo lambda – Inventado por Alonzo Church como un formalismo matemático para expresar el cómputo por medio de funciones.
Un cálculo que captura el aspecto computacional de la noción de función (Church [1932-1933, 1940])
Los lenguajes funcionales modernos pueden ser entendidos como un embellecimiento sintáctico de l.

Esos alumnos de Lenguajes formales, que es una función computable

Calculo Lambda

Alan Turing

- Diseñó una máquina hipotética (la que hoy conocemos como maquina de Turing) capaz de resolver cualquier función computable.
- Utilizó la maquina para demostrar que el problema de decisión (Entscheidungsproblem) era irresoluble.
- Llega a la conjetura que cualquier función computable es calculable por una máquina de Turing, el día de hoy conocemos esa conjetura como la tesis de Turing.

Calculo Lambda

Introducción

Alonzo Church

- Diseñó un sistema formal llamado calculo lambda el cual utilizó para demostrar que el problema de decisión (Entscheidungsproblem) era irresoluble.
- Propuso la definición de calculabilidad efectiva de como cualquier función que sea "λ-definible", dicho en otras palabras, que pueda ser programada en el calculo lambda, el día de hoy conocemos esa conjetura como la tesis de Church.

Calculo Lambda

Historia

- Alonzo Church y Alan Turing llegaron al mismo resultado pero de diferentes formas.
- Ya que ambas tesis, la de Turing y Church demuestran lo mismo, nos referimos a ellas como la tesis Church-Turing.

Función computable

Funciones que pueden ser calculadas con una máquina de cálculo.
Las funciones computables son usadas para discutir sobre computabilidad sin referirse a ningún modelo de computación concreto, como el de la máquina de Turing o el de la máquina de registros
Según la Tesis de Church-Turing, la clase de funciones computables es equivalente a la clase de funciones definidas por funciones recursivas, cálculo lambda, o algoritmos de Markov

Entiendo que una función computable es algo que se puede resolver por un algoritmo de computación, como calcular el consumo de combustible de un avión en determinadas condiciones, una función definible pero no computable seria entonces algo que se pueda definir, pero no sea sujeto a resolverse por una "receta de cocina" como los problemas de optimización de rutas (el problema del vendedor ambulante), juegos de intuición como el Go, o similares, donde el tiempo requerido para responder es mucho mayor a lo que las computadoras puedan ofrecer, o que estén sujetos a dinámicas no computables, como el clima.

El objetivo es dar una teoría general de las funciones es un sistema formal diseñado para definir funciones, la forma de utilizarlas y la recursión.
Es utilizado como un fundamento de lenguajes de programación porque aporta una sintaxis básica de programación, la semántica para el concepto e función en la transformación de argumentos en resultados y una forma de definir primitivas de programación.

El calculo lambda es el lenguaje de programación mas pequeño consiste en transformación simple(sustituir variables) y en definir funciones. El calculo lambda se puede decir que es equivalente a las máquinas Turing porque es capaz de evaluar y expresar cualquier función computable. Church había querido hacer un sistema formal completo para modelizar la matemática pero después separo el calculo lambda y lo ideo para que estudiara la computabilidad.

En el cálculo lambda, una expresión o término se define recursivamente a través de las siguientes reglas de formación:
Toda variable es un término: x, y, z, u, v, w, x1, x2, y9,... Si E1 es un término y x es una variable, entonces (λx.E1) es un término (llamado una abstracción lambda). Si E1 y E2 son términos, entonces (E1E2) es un término (llamado una aplicación lambda).

N

ADA MAS ES UN TERMINO

Formalidad

¿Para qué sirve el *Calculo Lambda* ?

- Modelo simple que permite representar todas las funciones computables.
- Marco formal para estudiar propiedades de procesos de cómputo, programas, lenguajes, entre otros
- Realización de pruebas de concepto para evaluar extensiones a lenguajes de programación.
- etc. (ver bibliografía).

Lambda

El concepto central en ***Calculo Lambda*** es la definición de su "expresión".

¡expresión¿ := ¡variable¿ — ¡abstracción¿ — ¡aplicación¿

¡abstracción¿ := λ ¡variable¿.¡expresión¿

¡aplicación¿ := ¡expresión¿¡expresión¿

Ejemplo

abstracción := λx.x
aplicación := (λx.x)y
expresión := (λx.x)((λx.x)y)

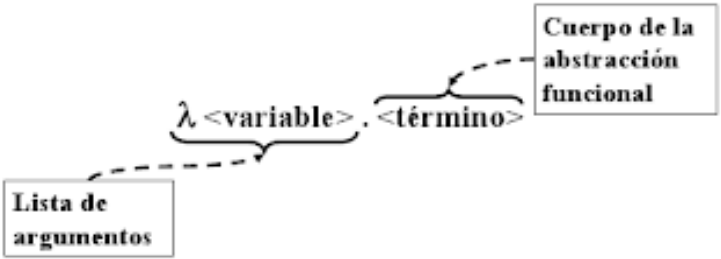
Calculo Lambda

En esta sintaxis no existe el concepto de ¡nombre¿ o ¡constante¿

¿Qué implica esto?—
El formalismo no tendrá primitivas, no nos permitirá emplear funciones con el concepto de módulos abstractos

Abstracción Funcional

Una abstracción lambda o abstracción funcional λx.t representa una función que toma solo a un argumento, además se dice que λ liga la variable x en el término t.



Términos lambda

v λv.E1 (E1 E2)
v es un nombre de variable, E1 Y E2 son lambda términos y λv.E1 se llaman abstracciones v es el parámetro formal y E1 es el cuerpo de la abstracción. λv.E1 es una función que recibe v como valor para sustituir y devuelve el valor de E1 para cada ocurrencia de (E1 E2) son llamados aplicaciones esta forma representa la llamada a la función E1 recibiendo como argumento E2.

Cadenas de ejemplos

x (xy) (((xz)y)x) (λx.x) ((λx.x)y) (λz.(λx.y)) ((x(λz.z))z)

Lambda Calculo

$(\lambda x{:}\text{Bool}.\ \lambda y{:}\text{Bool}\ !\ \text{Bool}.\ y\ (y\ x))((\lambda z : \text{Bool}.\ \text{true})\ \text{false})$

Calculo Lambda

Primera aproximación

Un primer vistazo a Lambda Calculus...

$0 = \lambda s.\lambda x.x$
$1 = \lambda s.\lambda x.s\ x$
$2 = \lambda s.\lambda x.s(s\ x)$
$3 = \lambda s.\lambda x.s(s(s\ x))$

Convenciones

Por convención se suelen omitir los paréntesis externos, ya que no cumplen ninguna función de desambiguación. Por ejemplo se escribe $(\lambda z.z)z$ en vez de $((\lambda z.z)z)$, y se escribe $x(y(zx))$ en vez de $(x(y(zx)))$. Además se suele adoptar la convención de que la aplicación de funciones es asociativa hacia la izquierda. Esto quiere decir, por ejemplo, que $xyzz$ debe entenderse como $(((xy)z)z)$, y que $(\lambda z.z)yzx$ debe entenderse como $((((\lambda z.z)y)z)x)$.

Las primeras dos reglas generan funciones, mientras que la última describe la aplicación de una función a un argumento. Una abstracción lambda λx.t representa una función anónima que toma un único argumento, y se dice que el signo λ liga la variable x en el término t. En cambio, una aplicación lambda ts representa la aplicación de un argumento s a una función t. Por ejemplo, λx.x representa la función identidad x $\rightarrow$ x, y (λx.x)y representa la función identidad aplicada a y. Luego, λx.y representa la función constante x $\rightarrow$ y, que develve y sin importar qué argumento se le dé.
Las expresiones lambda no son muy interesantes por sí mismas. Lo que las hace interesantes son las muchas nociones de equivalencia y reducción que pueden ser definidas sobre ellas.

Entonces

Para que todo esto?

Tiene por objeto explicitar el concepto que representa el empleo de funciones como medio de transformación de argumentos en resultados.

A este formalismo lo denominamos CÁLCULO, ya que el mismo empleará:

- un conjunto de axiomas, y
- reglas de inferencia (de la misma forma que lo utilizan los sistemas formales)

para representar el medio de transformación mencionado.

Calculo Lambda

El cálculo λ no tiene tipos (tomar en cuenta ámbito, paso de parámetros, estrategia de evaluación).

Programación funcional es esencialmente cálculo λ con constantes apropiadas.

El cálculo λ con tipos asocia un tipo con cada término.

Convenciones Sintácticas

Convenciones sintácticas para hacer más sencillas las λ- expresiones

1. La aplicación va a ser asociativa por la izquierda: (((MN)P)Q) $\rightarrow$MNPQ
2. La abstracción es asociativa por la derecha: (λx.(λy.M)) $\rightarrow$ λx.λy.M
3. La aplicación es prioritaria sobre la abstracción: (λx.(MN)) $\rightarrow$ λx.MN
4. Se puede suprimir símbolos λ en abstracciones consecutivas: λx. λy. ... λz.M $\rightarrow$ λxy ... z.M

Calculo Lambda

- El ***Calculo Lambda*** se puede ver como un lenguaje de definición de funciones cuyos únicos tipos de datos primitivos son las funciones
- (no hay números ni cadenas de caracteres), y que incluye un mecanismo de evaluación de las funciones basado en sustitución de variables por expresiones. En particular, todas las funciones tienen un argumento que representa a su vez otra función y sus imágenes son a su vez funciones.

Funciones λ

Ejemplos

- λx.x es la función identidad, que a cada función le hace corresponder ella misma
- x.y es la función con valor constante la función y λx.λy.x es la función que a cada función x le hace corresponder la función constante x
- λx.λy.y es la función constante identidad

Calculo Lambda

En ***Calculo Lambda*** las funciones son anónimas, se utiliza el símbolo lambda para definirlas y solo reciben un valor de entrada.
f(x) = x - y sería equivalente a λx.x - y f(x,y) = x - y sería equivalente a λx.λy.x - y

Reglas y Resolución

Estrategias de reducción.

Reglas de computación para seleccionar pasos de reducción .
Leftmost-Outermost (Normal order): El redex seleccionado en cada paso de reducción es aquel que comienza más a la izquierda en la expresión. (argumento no necesariamente en forma normal)
Leftmost-Innermost (Applicative order): El redex seleccionado en cada paso de reducción es aquel que termina más a la izquierda en la expresión. (la abstracción y argumento ya estarán en forma normal cuando se selecciona el redex).
Call-by-Name: LO pero no se reduce en el cuerpo de una abstracción.
Call-by-Value: LI pero no se reduce en el cuerpo de una abstracción.

Teorema del punto fijo. Toda expresión lambda e tiene un punto fijo e' tal que (e e') = e'.
Y = \f.(\x.f (x x)) (\x.f (x x)) (paradoxical combinator). (Y e) = e (Y e)
Toda función recursiva puede ser escrita sin usar recursión:
fac = Y (\fac.\n.if (n = 0) then 1 else (n * (fac (n - 1)))
Tesis de Church. Las funciones de N en N efectivamente computables son aquellas definibles en el cálculo lambda.
Kleene[1936]: l-definibilidad equivalente a recursividad (Gödel y Herbrand)
Turing[1937]: Turing-computabilidad equivalente a l-definibilidad.

True = \x.\y.x False = \x.\y.y cond = \p.\c.\a.p c a (cond p c a = if p then c else a)
(= n n) =¿ True (= n m) =¿ False
Que pasa con +, *, 0, 1? Alternativas: Codificación en l Nociones primitivas: semántica usando d-reducción
e := x — c — e1 e2 — \x.e
If True e1 e2 =¿ e1 If False e1 e2 =¿ e2 extensión conservativa: e1 ^{1} e2 en l entonces e1 ^{1} e2 en l d

Reglas de *Calculo Lambda*

- El cálculo debe proveer mecanismo por el cual se obtiene el resultado de aplicar una función a argumentos dados:
  M = N
  M y N son términos, = es una relación de equivalencia

Los axiomas y reglas de inferencia fijarán condiciones para términos equivalentes

Como resolver esto??

Para calcular el resultado de una aplicación lambda o funcional será atraves de la generación de expresiones equivalentes por la aplicación de las reglas del cálculo λ.

Sustitución

$$(\lambda x.x)(\lambda y.y) = x[x := (\lambda y.y)] = (\lambda y.y)$$
$$(\lambda x.(\lambda y.x + y))3 = (\lambda y.x + y)[x := 3] = (\lambda y.3 + y)$$

Otros

Booleanos
true := λx. λy.x
false := λx. λy.y
Aritmetica
suma := λm.λn.λf .λx. m f (n f x)
succ := λn . λf . λx . f (n f x)

Números

```
0 := $\lambda$f . $\lambda$x . x

1 := $\lambda$f . $\lambda$x . f x

2 := $\lambda$f . $\lambda$x . f (f x)

3 := $\lambda$f . $\lambda$x . f (f (f x))
```

((λx.λy.x)y)z -¿ ((λy.x)[y/x])z // substitute y for x in the body of "λy.x¿
((λy'.x)[y/x])z // after alpha reduction -¿ (λy'.y)z // first beta-reduction complete!
-¿ y[z/y'] // substitute z for y' in ɀ¿ y // second beta-reduction complete!

$$
\begin{array}{rl}
 & (\lambda xyz.xyz)(\lambda x.xx)(\lambda x.x)x \\
\to_\beta & (\lambda yz.xyz)[x := \lambda x.xx](\lambda x.x)x \\
\equiv & (\lambda yz.(\lambda x.xx)yz)(\lambda x.x)x \\
\to_\beta & (\lambda yz.(xx)[x := y]z)(\lambda x.x)x \\
\equiv & (\lambda yz.yyz)(\lambda x.x)x \\
\to_\beta & (\lambda z.yy)[y := \lambda x.x]x \\
\equiv & (\lambda z.(\lambda x.x)(\lambda x.x)z)x \\
\to_\beta & (\lambda z.x[x := \lambda x.x]z)x \\
\equiv & (\lambda z.(\lambda x.x)z)x \\
\to_\beta & (\lambda z.x[x := z])x \\
\equiv & (\lambda z.z)x \\
\to_\beta & z[z := x] \\
\equiv & x
\end{array}
$$

Valores Inmutables

Ahora que sabemos que los valores son inmutables ¿Por qué no lo usamos para eliminar las condiciones de carrera (race condition)? Veamos algunos casos importantes sobre este tema puntual...

Therac-25

T

herac-25 fue una maquina de radio terapia que al contar con una falla de software causó la muerte de 3 pacientes y otros quedaron con daños por sobredosis de radiación.

El diagnostico:

Se determinó que el mal funcionamiento del programa era ocasionado por una condición de carrera, relacionada con el desbordamiento de una variable contadora (Wikipedia).

Apagón del noreste del 2003

Se produjo un apagón que afectó buena parte de Estados Unidos y parte de Canadá.
El diagnostico:
Un bug en el software en el sistema de alarma en el cuarto de control. La falla en la alarma dejo a los trabajadores inadvertidos del problema, el bug fue una condición de carrera que puso la alarma en un bucle infinito. Wikipedia

Posible solución

Podemos utilizar programación funcional para cambiar el comportamiento anti natural del manejo de hilos con datos mutables y usar la naturaleza de la programación funcional para trabajar con datos inmutables.

Uso

Cálculo lambda y los lenguajes de programación

¿Qué tiene que ver esto conmigo. . . ?
Más de lo que imaginas.

Javascript

```
(function () {
  console.log("función anónima que se ejecuta automaticamente");
}());

var f = function(x) {
    console.log("función anónima guardada en una variable")
}

$(document).ready(function(){
    console.log("función que recibe otra funcion como parametro")
})
```

Python

```python
def mult(x, y):
    return x * y

def decorador_cuadrados(f):
    def cuadrados(x, y):
        return f(x**2, y**2)
    return cuadrados

@decorador_cuadrados
def nueva_mult(x, y):
    return x * y
```

```python
lista = [1, 2, 3]
nueva_lista = map(lambda x: x * 2, lista)
```

Definiciones

- Transparencia referencial, decimos que una expresión es transparentemente referencial si podemos reemplazar el valor de todas sus ocurrencias en el programa sin alterar el funcionamiento del mismo.
- Valores Inmutables, la modificación de los valores no existe, una vez creado ese valor no podrá mutar, lo que podemos hacer es crear nuevos valores a partir de los creados inicialmente.

Funciones de orden superior

Este tipo de funciones toman otras funciones como parámetros o retornan otras funciones como resultados.
Básicamente, son funciones que cumplen al menos una de las siguientes condiciones:

- Tomar una o más funciones como entrada
- Devolver una función como salida

```
def devuelve_funcion(cmd: String) = {
  cmd match {
    case "suma" => (x:Int, y:Int) => x + y
    case "multiplicacion" => (x:Int, y:Int) => x * y
  }
}

devuelve_funcion("suma")(1, 2)
```

```
def mult_2(x:Int) = x * 2

lista.map(mult_2(_))
```

Referencias

- R. Bird. , 2000, ***Introducción a la programación funcional con Haskell***, Prentice Hall.
- Graham Hutton, 2007, ***Programming in Haskell***, Cambridge University Press, New York, NY, USA.
- O'Sullivan, Bryan, Stewart, Don and Goerzen, John, 2008, ***Real World Haskell***, O'Reilly.
- B.C. Ruiz, F. Gutiérrez, P. Guerrero y J.E. Gallardo, 2004, ***Razonando con Haskell***, Thompson.
- S. Thompson , 1999, ***Haskell: The Craft of Functional Programming, Second Edition***, Addison-Wesley.