

EL CÁLCULO LAMBDA

λ



EL CÁLCULO LAMBDA λ



- El cálculo λ fue desarrollado por el matemático Alonzo Church en la década del 30, a fin de establecer una teoría general de funciones y extenderla a modo de proveer fundamentos para la lógica y matemática.
- Propiedades generales de las funciones independiente del área particular de problemas.



- Lambda ha sido empleado como fundamento de los lenguajes de programación, aportando:
 - Sintaxis básica
 - Semantica para el concepto defunción como proceso de transformación de argumentos en resultados
 - Medio para definir primitivas de programación

Computabilidad

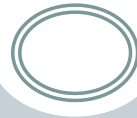


- **Algoritmo:** Procedimiento **sistemático** que permite resolver un problema en un número **finito** de pasos, cada uno de ellos especificado de manera **efectiva** y sin **ambigüedad**.
- **Función computable:** Aquella que puede ser calculada mediante un dispositivo **mecánico** dado un tiempo y espacio de almacenamiento **ilimitado** (pero finito)



- No importa la eficiencia, sino la posibilidad de ser calculada.
- ¿Existen funciones no computables?
- **Entscheidungsproblem**: Décima pregunta de Hilbert (Bologna, 1928): ¿Existe un procedimiento mecánico (algorítmico) general para resolver toda cuestión matemática bien definida?

Tesis Church-Turing



- Existen problemas bien definidos para los cuales no es posible encontrar un procedimiento mecánico que devuelva una solución en un tiempo finito.
 - **El problema de la detención**
 - El problema del castor laborioso (afanoso)
- **Tesis Church-Turing:** Toda función computable es calculable mediante una máquina de Turing.
 - Indemostrable, pero considerada cierta por la mayoría.
- Equivalencia entre distintos sistemas formales:
 - Máquina de Turing \leftrightarrow Cálculo lambda
 - Calculo lambda \leftrightarrow Funciones recursivas
 - etc.



- Simplificación extrema del cálculo:

- No importa el nombre de las funciones ni de los argumentos: $f(x,y) = x^2 + y^2$ y $g(a,b) = a^2 + b^2$ son la misma función.

$$(x, y) \rightarrow x^2 + y^2$$

- Toda función de más de un argumento se puede considerar como una función de **un solo** argumento que devuelve no un valor sino **una función:**
Currificación

$$x \rightarrow y \rightarrow x^2 + y^2$$

- No se necesitan números: Todo puede ser representado únicamente mediante **funciones**.



- Las máquinas de Turing, y sus extensiones en las máquinas RAM sirven de inspiración al **paradigma imperativo**:
 - Un cómputo es una secuencia de operaciones..
 - ..que **modifican el estado** del programa (registros)..
 - ..y cuyos resultados **determinan la secuencia** de ejecución.

Calculo Lambda



- Un cómputo El cálculo lambda (y su variante la lógica combinatoria) sirve de inspiración al **paradigma funcional**:
 - consiste en una **expresión** que puede ser transformada en otras mediante **reglas de reescritura**.
 - El orden de evaluación es **irrelevante**.
- Las máquinas de Turing, el cálculo lambda y las funciones recursivas son **equivalentes**.

EL CÁLCULO LAMBDA λ



- El cálculo λ puro, es una gramática para términos:

$M :: x \mid M_1 \ M_2 \mid (\lambda x. M)$

Variable = x, y, z, \dots

Término = M, N, P, Q (Un término puede ser una variable, una aplicación ($M \ N$) o una abstracción ($\lambda x. M$))

APORTE CONCEPTUAL A LOS LENGUAJES DE PROGRAMACIÓN



- Sintaxis de base.
- Una semántica para el concepto de función como proceso de transformación de argumentos en resultados.
- Medios para definir primitivas de programación.

$\langle \text{Termino} \rangle ::= \langle \text{Variable} \rangle$
 $| \lambda \langle \text{Variable} \rangle . \langle \text{Termino} \rangle$
 $| (\langle \text{Termino} \rangle \langle \text{Termino} \rangle)$



- En la sintaxis anterior no existe el concepto de

<nombre> o <constante>

Qué implica esto?

EL CÁLCULO LAMBDA λ



- 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.



- Una expresión lambda puede ser:
 - Una **variable** ($a, b, c \dots$)
 - Una **abstracción**: $\lambda x. t$ (donde x es una variable y t es una expresión lambda)
 - Una **aplicación**: $f \ g$ (donde f y g son expresiones lambda)

Abstracción Funcional o Lambda



$\lambda \text{ <variable> . <termino>}$

Notación para 0 o
más *conversiones* α
y *reducciones* β

Notación para 0 o
más *conversiones* α
y *reducciones* β

Una función que requiera más de un argumento se representa de la siguiente forma:

$\lambda x_1. \lambda x_2. \lambda x_3 \dots \lambda x_n. M$

Convenciones



- **Convenciones:**

- Las variables representan funciones.
- Se pueden usar paréntesis para indicar el orden de evaluación. $f\ g\ h = (f\ g)\ h$
- Las aplicaciones son asociativas hacia la izquierda:
- Las abstracciones se extienden todo lo posible hacia la derecha
- Las abstracciones se pueden **contraer**: $\lambda x. \lambda y. t \equiv \lambda x\ y. t$

Representación



- Representación de los números naturales, incremento, suma, producto, predecesor, resta:

$$\mathbf{0} \equiv \lambda f x . x$$

$$\mathbf{1} \equiv \lambda f x . f x$$

$$\mathbf{2} \equiv \lambda f x . f (f x)$$

$$\mathbf{3} \equiv \lambda f x . f (f (f x))$$

$$\mathbf{4} \equiv \lambda f x . f (f (f (f x)))$$

$$\mathbf{Succ} \equiv \lambda n f x . f (n f x)$$

$$\mathbf{Sum} \equiv \lambda m n . m \mathbf{Succ} n$$

$$\mathbf{Mul} \equiv \lambda m n . m (\mathbf{Sum} n) \mathbf{0}$$

$$\mathbf{Pred} \equiv \lambda n f x . n (\lambda g h . h (g f)) (\lambda n . x) (\lambda n . n)$$

$$\mathbf{Sub} \equiv \lambda n m . n \mathbf{Pred} m$$

Representación



- Representación de los valores lógicos, condicional, test si valor nulo, test menor o igual:

$$\mathbf{T} \equiv \lambda x y . x$$

$$\mathbf{F} \equiv \lambda x y . y$$

$$\mathbf{If} \equiv \lambda p a b . p a b$$

$$\mathbf{Is0} \equiv \lambda n . n (\lambda x . \mathbf{F}) \mathbf{T}$$

$$\mathbf{Leq} \equiv \lambda n m . \mathbf{Is0} (\mathbf{Sub} n m)$$

Representación



- El cálculo del máximo común divisor se puede expresar:

$$\mathbf{Mcd0} \equiv \lambda r a b . \mathbf{If} (\mathbf{Is0} b) a (\mathbf{If} (\mathbf{Leq} b a)$$

$$(r a (\mathbf{Sub} a b)) (r b (\mathbf{Sub} b a)))$$

$$\mathbf{Mcd} \equiv \lambda a b . \mathbf{Mcd0} (\mathbf{Y} \mathbf{Mcd0}) b a$$

- Expandiendo las definiciones:

$$\lambda a b . (\lambda r c d . (\lambda p e f . p e f) ((\lambda n . n (\lambda x g y . y) (\lambda x y . x)) d) c ((\lambda p h i . p h i) ((\lambda n m . (\lambda j . j (\lambda x k y . y) (\lambda x y . x)) ((\lambda l o . l (\lambda p f x . p (\lambda g h . h (g f)) (\lambda q . x) (\lambda s . s)) o) n m)) d c) (r c ((\lambda n m . n (\lambda t f x . t (\lambda g h . h (g f)) (\lambda u . x) (\lambda v . v)) m) c d)) (r d ((\lambda n m . n (\lambda w f x . w (\lambda g h . h (g f)) (\lambda y . x) (\lambda z . z)) m) d c)))) ((\lambda g . (\lambda x . g (x x)) (\lambda x . g (x x))) (\lambda r a' b' . (\lambda p c' d' . p c' d') ((\lambda n . n (\lambda x e' y . y) (\lambda x y . x)) b') a' ((\lambda p f' g' . p f' g') ((\lambda n m . (\lambda h' . h' (\lambda x i' y . y) (\lambda x y . x)) ((\lambda j' k' . j' (\lambda l' f x . l' (\lambda g h . h (g f)) (\lambda m' . x) (\lambda n' . n')) k') n m)) b' a') (r a' ((\lambda n m . n (\lambda o' f x . o' (\lambda g h . h (g f)) (\lambda p' . x) (\lambda q' . q')) m) a' b')) (r b' ((\lambda n m . n (\lambda r' f x . r' (\lambda g h . h (g f)) (\lambda s' . x) (\lambda t' . t')) m) b' a'))))) b a$$

REGLAS DEL CÁLCULO λ



- 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 terminos = es una relación de equivalencia

- Los axiomas y reglas de inferencia fijarán condiciones para términos equivalentes

REGLAS DEL CÁLCULO λ



- Para que relación $=$ sea efectivamente de equivalencia se consigue imponiendo las propiedades de:
 - Reflexividad $M = M$
 - Simetría $M=N$
 $N=M$
 - Transitividad $M = N$
 $N = P$ por lo tanto
 $M=P$

VARIABLES LIBRES Y ACOTADAS



- Definición (Ocurrencia Libres y Ligadas de variables)
 - Una ocurrencia de la variable x en un término P es ligada si y solo si aparece en un subtérmino de P de la forma $\lambda x.M$.
 - Cualquier otra ocurrencia de x en P es llamada libre.

Ej.:

$$((x \ v) \ \lambda y. (y \ v))w$$

- Las dos ocurrencias de y son ligadas.
- Todas las ocurrencias de x , v y w son libres.

VARIABLES LIBRES Y ACOTADAS



- Abstracciones $\lambda x. M$ restringen a x en $\lambda x. M$ o sea:
 - Variable x acotada en $\lambda x. M$.
 - El conjunto *libre* (m) formado por variables libres de M .

- Reglas de sintaxis:

$$\text{Libre}(x) = \{x\}$$

$$\text{Libre}(MN) = \text{libre}(M) \cup \text{libre}(N)$$

$$\text{Libre } \lambda x. M = \text{libre}(M) - \{x\}$$

SUSTITUCIÓN



La sustitución por un término N de una variable x en M se escribe $(N/x)M$ y se define:

- Suponga que las variables libres de N no tienen apariciones acotadas en M , entonces el término $(N/x)M$ se forma reemplazando con N todas las apariciones. libres de x en M .
- Otro caso suponga que la variable y es libre en N y acotada en M . La asociación y las apariciones acotadas correspondiente de y en M se reemplazan de manera consistente por una variable nueva z . Se sigue de nuevo las variables acotadas en M hasta que se pueda aplicar el caso anterior.

SUSTITUCIÓN



- En los siguientes casos, M no tiene apariciones acotadas, así N reemplaza todas las apariciones de x en M para formar $(N/x) M$:

$$(u/x) x = u$$

$$(u/x) (x x) = (u u)$$

$$(u/x) (x y) = (u y)$$

$$(u/x) (x u) = (u u)$$

$$((\lambda x. x)/x)x = (\lambda X. x)$$

SUSTITUCIÓN



- En lo sig. casos M no tiene apariciones libres de x , así que $(N/x) M$ es M mismo:

$$(u/x) y = y$$

$$(u/x) (y z) = (y z)$$

$$(u/x) (Ay y) = (Ay y)$$

$$(u/x) (AX x) = (AX x)$$

$$((\lambda x. x)/x)y = y$$

SUSTITUCIÓN



- En lo sig. casos la variable u en M tiene apariciones acotadas en M de modo que $(N/x) M$ se forma primero asignando nuevo nombre, a las apariciones acotadas de u en M :

$$\{u/x\} (\lambda u. x) = \{u/x\} (\lambda z. x) = (\lambda z. u)$$

$$\{u/x\} (\lambda u. u) = \{u/x\} (\lambda z. z) = (\lambda z. z)$$

SUSTITUCIÓN



- Ejemplos:

- $[N/x]x ::= N$

- $[N/x]V ::= V$

- ✦ si $V \neq x$

- $[N/x](PQ) ::= ([N/x]P \quad [N/x]Q)$

- $[N/x]\lambda x. M ::= \lambda x. M$

- $[N/x]\lambda y. P ::= \lambda y. [N/x]P$

- ✦ Si y no ocurre libre en N o x no ocurre libre en P

- $[N/x]\lambda y. P ::= \lambda z. [N/x][z/y]P$

- ✦ Si Y no ocurre libre en N y X ocurre libre en P y Z no ocurre libre en P

Reglas de manipulación



- La **α-reducción** (renombrado): Es posible cambiar el nombre de las variables ligadas.

$$\lambda x. x y \equiv \lambda a. a y$$

- La **β-reducción**: Al **aplicar** una abstracción a otra expresión, podemos sustituir la expresión por el término de la abstracción donde se han **sustituido** todas las apariciones de la variable ligada por la expresión aplicada:

$$(\lambda x. x (y x)) (z w) \equiv (z w) (y (z w))$$

- La **η-reducción**: Si el término de una abstracción es una aplicación donde en la primera expresión no aparece la variable ligada y la segunda expresión es la variable, se puede sustituir la abstracción por la expresión:

$$\lambda x. f x \equiv f$$

Reducción Beta



- El aspecto central del cálculo consiste en establecer el mecanismo por el cual se obtiene el resultado de aplicar una función (representada por una abstracción) a un argumento (representado por un término cualquiera).

Ej.:

abstracción funcional = $(x)\text{sqrt}(2 * x + 1)$

aplicado a un valor = $(x)\text{sqrt}(2 * x + 1)(4)$

se puede reducir a = $(x)\text{sqrt}(2 * 4 + 1)$

AXIOMAS Y REGLAS DE LA IGUALDAD BETA



- Axioma fundamental: $(\lambda x.M)N =_{\beta} \{N/x\} M$ (axioma β)
Así $(\lambda x.x) u =_{\beta} u$ y $(\lambda x.y) u =_{\beta} y$.

$(\lambda x.M) =_{\beta} \lambda z.\{z/x\}M$ suponiendo que z no es libre en M
(axioma α)

$$M =_{\beta} M \quad (\text{regla de reflexividad})$$

$$\frac{M =_{\beta} N}{N =_{\beta} M} \quad (\text{regla de conmutatividad})$$

AXIOMAS Y REGLAS DE LA IGUALDAD BETA



$$\frac{M =_{\beta} N \quad N =_{\beta} P}{N =_{\beta} P} \text{ (regla de Transitividad)}$$

$$\frac{M =_{\beta} M' \quad N =_{\beta} N'}{MN =_{\beta} M'N'} \text{ (regla de congruencia)}$$

$$\frac{M =_{\beta} M'}{\lambda x. M =_{\beta} \lambda x. M'} \text{ (regla de congruencia)}$$

- El efecto de un programa funcional es el cómputo de un valor, imagen en la función representada por el programa de un cierto valor dado.

REDUCCIONES (redex)



- Redex representa la idea de un computo que esta por realizarse.
- Un redex representaría el hecho de aplicar un programa a cierto dato y su forma normal.
- El proceso por el cual se obtiene esta forma normal será llamado reducción y podría verse como la «Ejecución» del «Programa» para el «Dato» en cuestión.


EJEMPLOS



EJEMPLOS



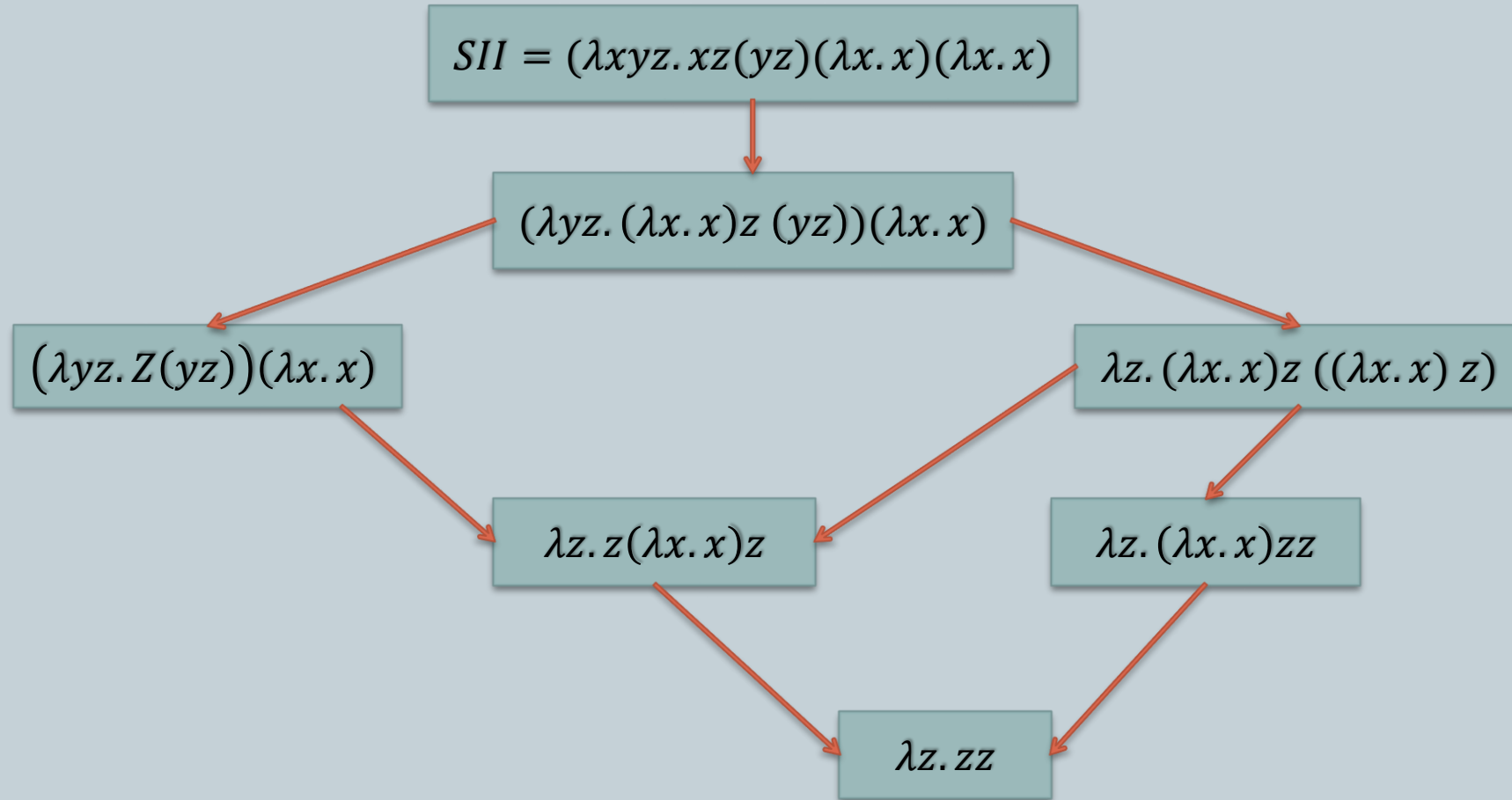
$$\begin{aligned} & (\lambda x y z . x y z)(\lambda x . x x)(\lambda x . x) x \\ \rightarrow_{\beta} & (\lambda y z . x y z)[x := \lambda x . x x](\lambda x . x) x \\ \equiv & (\lambda y z . (\lambda x . x x) y z)(\lambda x . x) x \\ \rightarrow_{\beta} & (\lambda y z . (x x)[x := y] z)(\lambda x . x) x \\ \equiv & (\lambda y z . y y z)(\lambda x . x) x \end{aligned}$$


$$\begin{aligned} \rightarrow_{\beta} & (\lambda z . y y)[y := \lambda x . x] x \\ \equiv & (\lambda z . (\lambda x . x)(\lambda x . x) z) x \\ \rightarrow_{\beta} & (\lambda z . x[x := \lambda x . x] z) x \\ \equiv & (\lambda z . (\lambda x . x) z) x \\ \rightarrow_{\beta} & (\lambda z . x[x := z]) x \\ \equiv & (\lambda z . z) x \\ \rightarrow_{\beta} & z[z := x] \\ \equiv & x \end{aligned}$$

REDUCCIONES (redex)




• Ej.:



TEOREMA DE CHURCH-ROSSER

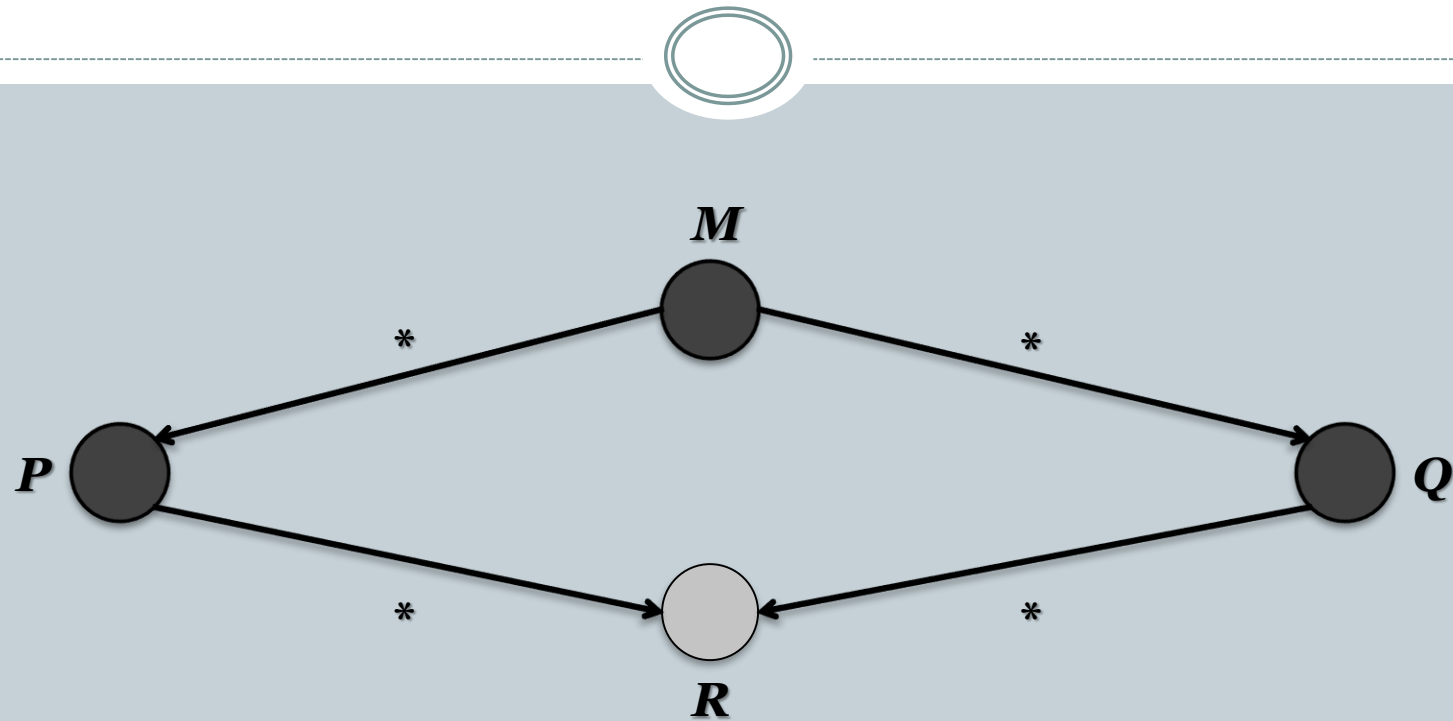


- Para todos los términos A puros M, P y Q , si $M \Rightarrow P$ y $M \Rightarrow^* Q$, entonces debe existir un término R tal que $P \Rightarrow^* R$ y que $Q \Rightarrow^* R$

A red arrow pointing from the text box to the \Rightarrow^* notation in the theorem statement.

Notación para 0 o
más *conversiones* α
y *reducciones* β

TEOREMA DE CHURCH-ROSSER




- Si M se reduce a P y a Q , entonces ambos pueden llegar a algún R común.

REGLAS DE CÁLCULO



- En los lenguajes de programación las aplicaciones $M\ N$ de funciones, se dice que se invocan por valor.
- Para cálculo lambda estrategia de reducción es una regla para escoger *redex*. Hace corresponder a cada término P (que no esté en forma normal), con un término Q tal que $P \Rightarrow_{\beta} Q$.
 - Reducción para las invocaciones por nombre elige la redex del extremo izquierdo y más externa.
 - Reducción para las invocaciones por valor elige la redex del extremo izquierdo y más interna de un término.

- 
- $(\lambda pq.pqp)(\lambda ab.a)(\lambda ab.b)$

Invocación por nombre : $(\lambda p q. p q p)(\lambda a b. a)(\lambda a b. b)$



Estrategia: Redex más a la izquierda, más externo primero, pero sin reducción bajo lambda.

- $(\lambda p q. p q p)((\lambda a b. a)(\lambda c d. d)) \rightarrow$
- $\lambda q. ((\lambda a b. a)(\lambda c d. d)) q ((\lambda a b. a)(\lambda c d. d)) \rightarrow$
- $\lambda q. ((\lambda a b. a)(\lambda c d. d)) q ((\lambda a b. a)(\lambda c d. d)) \rightarrow$
- $\lambda q. (\lambda b. (\lambda c d. d)) q ((\lambda a b. a)(\lambda c d. d)) \rightarrow$
- $\lambda q. (\lambda c d. d) ((\lambda a b. a)(\lambda c d. d)) \rightarrow$
- $\lambda q. (\lambda d. d)$

Invocación por valor: $(\lambda pq.pqp)(\lambda ab.a)(\lambda ab.b)$



Estrategia: más a la derecha, redex más interno primero pero sin reducción bajo lambda

- $(\lambda pq.pqp)((\lambda ab.a)(\lambda cd.d)) \rightarrow$
- $(\lambda pq.pqp)(\lambda b.(\lambda cd.d)) \rightarrow$
- $(\lambda pq.pqp)(\lambda b.(\lambda cd.d)) \rightarrow$
- $\lambda q.(\lambda bcd.d)q(\lambda bcd.d) \rightarrow$
- $\lambda q.(\lambda cd.d)(\lambda bcd.d) \rightarrow$
- $\lambda q.(\lambda d.d)$

Ejercicios



- Reducir las siguientes expresiones por valor y por nombre hasta llegar a una forma normal.
 - $((\lambda x. \lambda y. y \ x) \ z) \ v$
 - $((\lambda u. v \ (\lambda x. ((x \ x) \ y) \ (\lambda x. ((x \ x) \ y))))$

REGLAS DE CÁLCULO



- A pesar de que ocurra una evaluación sin final, los lenguajes funcionales utilizan la invocación por valor debido a que es posible implantarla eficientemente y alcanza la forma normal con la frecuencia necesaria.

Ejemplos



```
(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")  
})
```

Ejemplos



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