Lenguajes funcionales: λ-cálculo

- **λ-cálculo** (Church 1933)
  - Cálculo para el estudio formal del comportamiento de las funciones
    - Sintaxis: λ–expresiones
    - Reglas de reducción de λ–expresiones
  - Método matemático simple y consistente
  - Es un Modelo Computacional: *"todas las funciones computables son expresables en el λ-cálculo"*

- **λ-cálculo y programación funcional**

Los lenguajes funcionales modernos (Haskell, Miranda) son enriquecimientos (no triviales) del λ-cálculo con tipos.

Características provenientes del λ-cálculo:

- Funciones como ciudadanos de primera clase
  - Funciones de orden superior (currificación)
  - Funciones anónimas (λ-notación)
- Semántica operacional
  - Evaluación de expresiones (funcionales) por reducción

Características provenientes del λ-cálculo con tipos:

- Tipado fuerte
- Inferencia de tipos
- Polimorfismo

Implementación:

El λ-cálculo puede usarse como lenguaje intermedio en el proceso de implementación de un lenguaje funcional:

Haskell
⇓ - - - - - - - traducción a código intermedio
λ-cálculo
⇓ - - - - - - - implementación
código

El lenguaje del λ-cálculo (sin tipos)

- Sintaxis:

<expresión> ::=

<variable> |	x, y, .., f, g ..
<constante> |	0,1, true, +,...
λ<variable>.<expresión> |	λ-abstracción
<expresión> <expresión>	aplicación

- Convenios sintácticos:
  - "aplicación" más prioritaria que "λ-abstracción"

  Ej: λx. x y significa λx.(x y) y no (λx. x) y
  - "aplicación" asociativa a izquierdas

  Ej: x y z significa (x y) z y no x (y z)

- Funciones predefinidas:

  Hemos definido el λ-cálculo con constantes:

  números y operaciones aritméticas,

  booleanos y conectivos lógicos,

  operaciones sobre listas (cons, head, tail), ...

  razonable a efectos prácticos, PERO

  podríamos haber definido el λ-cálculo puro (sin <constante>) ya que cada constante se puede definir mediante una λ-abstracción.

  Ej: true ≡ λx. λy. x

- Currificación:

  Consideramos la λ-abstracción con un único argumento

  Ej: $\lambda x_1,x_2,...,x_n \,.\, e \equiv \lambda x_1.\, \lambda x_2.\, ...\, x_n \,.\, e$

Ejemplos de λ-expresiones

1. λx. + x 1 "la función de x que suma a x el número 1"
2. λx. λy. + x y función "suma" (curry)
3. λx. true función de x que devuelve true
4. λf. λx. f(f x) función "doble"
5. (λx. λy. + x y) 1 aplicación parcial de "suma" a 1
6. (λx. λy. + x y) 1 3 aplicación de "suma" a 1 y 3
7. (λy. + 1 y)
8. (λf. λx. f(f x)) (λy. + 1 y)
9. (λf. λx. f(f x)) (λy. + 1 y) 6

➢ ¿Qué expresiones son las tres últimas?

➢ ¿Cómo se escriben todas ellas en Haskell?

- **Aplicación**: aspecto computacional de las funciones

  función argum.

  (λ x . e) e' ➔ **e [e'/ x]**

  parámetro formal — cuerpo — se reduce a — resultado

  "El resultado de aplicar una λ-abstracción a un argumento es una instancia del cuerpo obtenida al sustituir las apariciones (libres) del parámetro formal por el argumento".

➢ Necesitamos definir formalmente la operación de sustitución

Ejemplos:

- (λx. ÷ x 2) 6 ➔ ÷ 6 2 ➔ 3
- (λx. λy. ÷ x y) 6 2 ➔ (λy. ÷ 6 y) 2 ➔ ÷ 6 2 ➔ 3
- (λf. λx. f x) (λy. ÷ y 2) 6
  - ➔ (λx. (λy. ÷ y 2) x) 6
  - ➔ (λy. ÷ y 2) 6
  - ➔ ÷ 6 2
  - ➔ 3

Ejercicio: Reducir la λ-expresión (λx. (λx. +(– x 1)) x 3) 9

- **Sustitución**: el problema de la "captura del nombre"

1. sólo se sustituyen apariciones libres de y:
   $(\lambda y.\ y(u(\lambda y.\ u\ y)))\ (\lambda x.\ x\ a) \rightarrow$
   $y(u(\lambda y.\ u\ y))\ [\lambda x.\ x\ a\ /\ y] \equiv (\lambda x.\ x\ a)\ (u(\lambda y.\ u\ y))$

2. captura de nombre:
   e **e'**

$(\ \lambda y.\ \lambda a.\ a\ y\ )\ (\lambda x.\ x\ a) \rightarrow (\lambda a.\ a\ y)\ [\lambda x.\ x\ a\ /\ y] \neq \lambda a.\ a\ (\lambda x.\ x\ a)$

Problema: **a** libre en **e'** ha quedado capturada por **λa** de **e**
Solución: aplicar α-conversión $(\lambda a.\ a\ y \equiv \lambda b.\ b\ y)$ de forma que
$(\lambda a.\ a\ y)\ [\lambda x.\ x\ a\ /\ y] \equiv (\lambda b.\ b\ y)\ [\lambda x.\ x\ a\ /\ y] \equiv \lambda b.\ b\ (\lambda x.\ x\ a)$

- **Variables libres en λ-expresiones**

Lib(e) = conjunto de variables libres (al menos 1 aparición) en **e**

- Lib(x) = {x} (x var)
- Lib(c) = {} (c cte)
- Lib(λx. e) = Lib(e) \ {x} (\ diferencia de conjs.)
- $Lib(e_1\ e_2) = Lib(e_1) \cup Lib(e_2)$

Ejemplo: Lib((λx. λy. y x) ((λz. z x) x)) = {x}

apar. ligadas apar. libres

En $(\lambda a.\ a\ y)\ [\lambda x.\ x\ a\ /\ y]$ la captura de nombre se dio porque
$a \in Lib(\lambda x.\ x\ a)$ y además
$y \in Lib(a\ y)$

- **La operación de Sustitución**

e [e'/ y] = sustitución de (todas las apariciones libres) de **y** por **e'** en **e**

- $x\ [e'/y] = x$, si $x \neq y$
  $y\ [e'/y] = e'$
- $c\ [e'/y] = c$ (c cte)
- $(e_1\ e_2)\ [e'/y] = (e_1\ [e'/y])(e_2\ [e'/y])$
- $(\lambda x.\ e)\ [e'/y]$
  $= \lambda x.\ e$ si $x = y$
  $= \lambda x.\ (e\ [e'/y])$ si $x \neq y \wedge (x \notin Lib(e') \vee y \notin Lib(e))$
  $= (\lambda z.\ (e\ [z/x]))\ [e'/y]$ si $x \neq y \wedge x \in Lib(e') \wedge y \in Lib(e)$

aplicada **α-reducción**: $(\lambda x.\ e) \rightarrow \lambda z.\ (e\ [z/x])$ con var nueva **z**
(renombramos **x** por **z**)

- **Reglas de reducción de expresiones**

β-reducción: $(\lambda x.\ e)\ e' \rightarrow e\ [e'/x]$

β-redex (con **α-reducción** si necesario)

$e \rightarrow_\beta e'$ sii **e'** se obtiene reduciendo una subexpresión de **e** que es un β-redex

$e \rightarrow^*_\beta e'$ sii existen $e_1, e_2, ..., e_{n-1}$ $(n>0)$ tales que
$e_1 \rightarrow_\beta e_2 \rightarrow_\beta ..... \rightarrow_\beta e_{n-1} \rightarrow_\beta e'$

δ-reducción: reducción para las constantes predefinidas
(+, ÷, *if-then-else*, *head*, *tail*, ...)

Ejs.: $-\ 5\ 4 \rightarrow_\delta 1$ $-\ 5\ (\div\ 6\ 2) \rightarrow_\delta -\ 5\ 3 \rightarrow_\delta 2$
if true *then* e *else* e' $\rightarrow_\delta$ e *tail*(*cons* e e') $\rightarrow_\delta$ e'

• Forma normal de una λ-expresión

La forma normal de una expresión **e** *es otra expresión* **e'** *tal que*

e $\rightarrow^*$ **e'** (con $\rightarrow$ siendo $\rightarrow_\beta$ ó $\rightarrow_\delta$) y

e' no tiene ningún β-redex ó δ-redex.

es decir, **e'** *es el resultado final del cómputo de* **e**

- No toda expresión tiene forma normal: *"hay computaciones que no terminan"* (λx. x x) (λx. x x)
- La forma normal (si existe) es única (módulo α-conversión): *"todos los cómputos de* **e** *que terminan obtienen el mismo resultado"*

(λx. f x) ((λy. g y) a) $\rightarrow$ f ((λy. g y) a) $\rightarrow$ f(g a)

(λx. f x) ((λy. g y) a) $\rightarrow$ (λx. f x) (g a) $\rightarrow$ f(g a)

• Ordenes (o estrategias) de reducción

Orden Aplicativo

- "leftmost innermost" redex
- Trata de reducir los argumentos antes de aplicar la función
- Evaluación impaciente
- Puede no encontrar la forma normal de una expresión:

(λx. z) ((λx. x x) (λx. x x)) $\rightarrow^*$ (λx. z) ((λx. x x) (λx. x x))

Orden Normal

- "leftmost outermost" redex
- Trata de reducir la función sin reducir los argumentos
- Evaluación perezosa (O.N + no repeticiones)
- Siempre obtiene la forma normal de una expresión (si existe)

(λx. z) ((λx. x x) (λx. x x)) $\rightarrow$ z

SEMANTICA OPERACIONAL del λ-cálculo

- Da significado a las λ-expresiones en base a cómo se opera con ellas
- Un programa funcional (por ejemplo en Haskell) junto con la expresión a evaluar (algoritmo + input) puede ser representado por (o traducido a) una λ-expresión **e**
- Computación: consiste en reducir **e** hasta su forma normal

La semántica operacional se basa en dos conceptos:

Reglas de reducción: β-reducción (con α-reducción implícita en la sustitución) y δ-reducción

Orden de reducción: orden normal

(En la evaluación perezosa de Haskell $\rightarrow$ "reducción de grafos")

SEMANTICA DENOTACIONAL del λ-cálculo

Dominios sintácticos:

Ψ: Pro	programas
E: Exp	λ-expresiones
V: Var	variables
C: Cte	constantes predefinidas

Reglas de producción abstractas:

Ψ ::= E

E ::= V | C | E_1 E_2 | λV.E

Dominios semánticos:

d : D D = dominio de valores (incluye funciones : D → D)

σ : S = Var → D valoración de variables

Funciones semánticas:

$\mathcal{M}$: Pro $\rightarrow$ D

$\mathcal{E}$: Exp $\rightarrow$ (S $\rightarrow$ D)

Ecuaciones semánticas:

$\mathcal{M}$ [[Ψ]] = $\mathcal{E}$ [[Ψ]] σ_0 siendo σ_0 : Var $\rightarrow$ D

V $\mapsto$ $\perp$

$\mathcal{E}$ [[V]] σ = σ [[V]]

$\mathcal{E}$ [[C]] σ = C

$\mathcal{E}$ [[E_1 E_2]] σ = ($\mathcal{E}$ [[E_1]] σ) ($\mathcal{E}$ [[E_2]] σ)

(aplicación de una función ∈ D a un argumento ∈ D)

$\mathcal{E}$ [[λV.E]] σ = f_E : D $\rightarrow$ D definida como

f_E (d) = $\mathcal{E}$ [[E]] (σ [d/V]) , para todo d∈ D

Ejercicio

Dar el significado denotacional del siguiente programa funcional:

Ψ = (λx.λy. x) (λu. 3) (λz. z)

Solución:

Sea σ_0 : V $\mapsto$ $\perp$ (valoración inicial)

$\mathcal{M}$ [[Ψ]] = $\mathcal{E}$ [[Ψ]] σ_0 =

($\mathcal{E}$ [[(λx.λy. x) (λu. 3)]] σ_0) ($\mathcal{E}$ [[λz. z]] σ_0) =

(($\mathcal{E}$ [[λx.λy. x]] σ_0) ($\mathcal{E}$ [[λu. 3]] σ_0)) ($\mathcal{E}$ [[λz. z]] σ_0) = (*)

Ahora veamos cada parte:

- $\mathcal{E}$ [[λz. z]] σ_0 = ident : D $\rightarrow$ D
  siendo ident (d) = $\mathcal{E}$ [[z]] (σ_0[d/z])
  = (σ_0[d/z]) [[z]]
  = d

- $\mathcal{E}$ [[λu. 3]] σ_0 = tres : D $\rightarrow$ D
  siendo tres (d) = $\mathcal{E}$ [[3]] (σ_0[d/u]) = 3
- $\mathcal{E}$ [[λx. λy. x]] σ_0 = prim : D $\rightarrow$ D
  siendo prim (d) = $\mathcal{E}$ [[λy. x]] (σ_0[d/x])

Entonces:

(*) = (prim tres) ident = ($\mathcal{E}$ [[λy. x]] (σ_0[tres/x])) ident = (**)

Ahora bien:

- $\mathcal{E}$ [[λy. x]] (σ_0[tres/x]) = f : D $\rightarrow$ D
  siendo f (d) = $\mathcal{E}$ [[x]] ((σ_0[tres/x])[d/y])
  = ((σ_0[tres/x])[d/y]) [[x]] = tres

Entonces: (**) = f ident = tres

y $\mathcal{M}$ [[Ψ]] = tres siendo tres(d) = 3

PODER EXPRESIVO del λ-cálculo

El λ-cálculo permite expresar todas las funciones computables.

- Valores booleanos y condicional:
  true = λx.λy.x false = λx.λy.y
  if b then e else e' = b e e'
- Los números naturales (numerales de Church):
  0 = λf.λx.x
  n = λf.λx.f(f(....(f x)...)) -- con n f 's
  suc = λx.λy.λz.y(x y z)
- Similarmente, se pueden dar λ-expresiones para las tuplas y sus proyecciones, las listas y sus operaciones, etc.

Ejercicio: Demostrar que if true then e else e' $\rightarrow^*_\beta$ e y que suc k $\rightarrow^*_\beta$ k+1

La recursión y el operador Y

La recursión se expresa mediante el operador de punto fijo Y.

- Para toda λ-expresión **e** existe otra **e'** tal que **e e'** $=_\beta$ **e'** (teorema del punto fijo), siendo $=_\beta$ la relación de equivalencia generada por $\leftarrow\rightarrow^*_\beta$
- El operador (o combinador) de punto fijo

  $Y = \lambda f.(\lambda x.f\ (x\ x))(\lambda x.f\ (x\ x))$

  expresable en el λ-cálculo, obtiene el menor punto fijo de cualquier λ-expresión **e**, es decir: **e (Y e)** $=_\beta$ **Y e**
- El significado de una función recursiva **f** es el menor punto fijo de un funcional **F**, y por tanto se expresa como **Y F**

Ejemplo: Definición de la función recursiva "factorial"

¿Cómo expresar la función factorial mediante una λ-expresión Fact? Deberá verificar la ecuación:

$$Fact =_\beta \lambda x.\ \text{if } x = 0 \text{ then } 1 \text{ else } x * Fact\ (x-1)$$

o equivalentemente:

$$Fact =_\beta (\lambda f.\lambda x.\ \text{if } x = 0 \text{ then } 1 \text{ else } x * f\ (x-1))\ Fact$$

es decir, Fact es el (menor) punto fijo del funcional F:

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

Por tanto puede expresarse como Y F:

$$Fact = Y\ (\lambda f.\lambda x.\ \text{if } x = 0 \text{ then } 1 \text{ else } x * f\ (x-1))$$