Llenguatges de Programació

Fonaments: λ-càlcul

Albert Rubio, Jordi Petit, Fernando Orejas

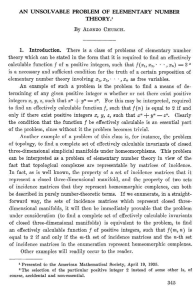
λ

Universitat Politècnica de Catalunya, 2019

Introducció

El **λ-càlcul** és un model de computació funcional, l'origen dels llenguatges funcionals, i la base de la seva implementació.

Inventat per Alonzo Church, cap al 1930.


AN UNSOLVABLE PROBLEM OF ELEMENTARY NUMBER THEORY.

By Alonzo Church.

Consisteix en agafar una línia de símbols i aplicar una operació de *cut-and-paste*.

$$(\lambda y.x(yz))\ (ab)$$

$$\Downarrow$$

$$x(abz)$$

Vídeo: Math whizzes of ancient Babylon figured out forerunner of calculus

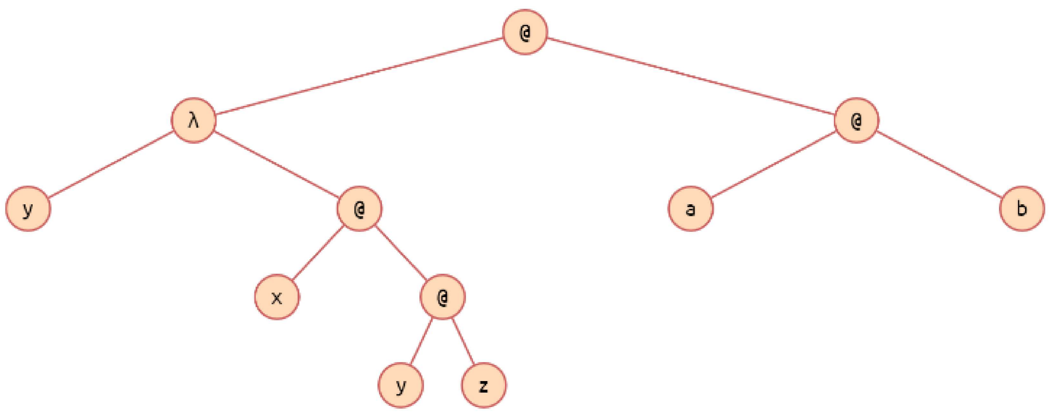
Fotos: Fair Use, jstor.org, Lambda Calculus for Absolute Dummies

Gramàtica

$$
\begin{aligned}
\textit{terme} &:= \textit{lletra} \mid (\ \textit{terme}\ ) \mid \textit{abstracció} \mid \textit{aplicació} \\
\textit{abstracció} &:= \boldsymbol{\lambda}\ \textit{lletra}\ .\ \textit{terme} \\
\textit{aplicació} &:= \textit{terme}\ \textit{terme}
\end{aligned}
$$

Exemples de termes:

- x
- $\lambda x.\, x$
- $(\lambda y.\, x(yz))(ab)$



Gramàtica

Les lletres es diuen *variables* i no tenen cap significat. El seu nom no importa. Si dues variables tenen el mateix nom, són la mateixa cosa.

Els parèntesis agrupen termes. Per claredat, s'agrupen per l'esquerra:

$$abcd \equiv (((ab)c)d).$$

La λ amb el punt introdueix funcions. Per claredat, es poden agrupar λs:

$$\lambda x.\, \lambda y.\, a \equiv \lambda x.\, (\lambda y.\, a) \equiv \lambda xy.\, a$$

Operacions

Només hi ha dues operacions per la construcció de termes:

- L'**abstracció** captura la idea de definir una funció amb un paràmetre:

  $$\lambda x.\, u$$

  on u és un terme.

  Diem que λx és el *cap* i que u és el *cos*.

  Intuició: $f(x,y) = x^2 + 2y + x - 1$ és representat per $\lambda x.\, \lambda y.\, x^2 + 2y + x - 1$

- L'**aplicació** captura la idea d'aplicar una funció sobre un paràmetre:

  $$f\ x$$

  on f i x són dos termes.

Funcions al curry

Al λ-càlcul totes les funcions tenen un sol paràmetre.

Les funcions que normalment consideraríem que tenen més d'un paràmetre es representen com a funcions d'un sol paràmetre utilitzant la tècnica del *currying*:

- Una funció amb dos paràmetres, com ara la suma, +: int x int ⟶ int, es pot considerar equivalent a una funció d'un sol paràmetre que retorna una funció d'un paràmetre, +: int ⟶ (int ⟶ int).
- Això vol dir que $2 + 3$, amb notació prefixa $(+2\ 3)$, s'interpretaria com $(+2)\ 3$, on $(+2)$ és la funció que aplicada a qualsevol paràmetre x, retorna $x + 2$.

Computació

La **β-reducció** (*cut-and-paste*) és la regla essencial de computació del λ-càlcul:

$$(\lambda x.\, u\; v) \longrightarrow_\beta u[x := v]$$

on $u[x := v]$ vol dir reescriure u substituint les seves x per v.

Exemple: $(\lambda y.\, x(yz))(ab) \longrightarrow_\beta x((ab)z) \;\equiv\; x(abz)$.

Si una expressió no pot β-reduir-se, aleshores es diu que està en **forma normal**.

Si $t \longrightarrow \ldots \longrightarrow t'$ i t' està en forma normal, aleshores es diu que t' és la forma normal de t, i es considera que t' es el resultat de l'avaluació de t.

Una λ-expressió té, com a màxim, una forma normal.

Variables lliures i lligades

Dins d'un terme, una variable és **lligada** si apareix al cap d'una funció que la conté. Altrament és **lliure**.

Les variables poden ser lliures i lligades alhora en un mateix terme.

Per exemple:

$$(\lambda x.\, xy)(\lambda y.\, y)$$

- y és lliure a la primera subexpressió.
- y és lligada a la segona subexpressió.

El problema de la captura de noms

Quan s'aplica la β-reducció s'ha de tenir cura amb els noms de les variables i, si cal, renomenar-les.

El problema es pot veure en el següent exemple: Sigui TWICE:

$$\lambda f.\, \lambda x.\, f(fx)$$

Calculem (TWICE TWICE):

$$\begin{aligned} \text{TWICE TWICE} &= (\lambda f.\, \lambda x.\, f(fx))\text{TWICE} \\ &\longrightarrow_\beta (\lambda x.\, \text{TWICE}(\text{TWICE}\ x)) \\ &= (\lambda x.\, \text{TWICE}(\lambda f.\, \lambda x.\, f(fx))x) \end{aligned}$$

Aplicant la β-reducció directament tindríem:

$(\lambda x.\, \text{TWICE}(\lambda f.\, \lambda x.\, f(fx))x) \longrightarrow_\beta (\lambda x.\, \text{TWICE}(\lambda x.\, x(xx)))$ **ERROR**

El que hauríem de fer és renomenar la variable lligada x mes interna:

$(\lambda x.\, \text{TWICE}((\lambda f.\, \lambda x.\, f(fx))x) = (\lambda x.\, \text{TWICE}((\lambda f.\, \lambda y.\, f(fy))x)$
$\longrightarrow_\beta (\lambda x.\, \text{TWICE}((\lambda y.\, x(xy))$ **OK**

α-Conversió

A més de la β-reducció, al λ-càlcul tenim la regla de l'α-conversió per renomenar les variables. Per exemple:

$$\lambda x.\, \lambda y.\, xy \longrightarrow_a \lambda z.\, \lambda y.\, zy \longrightarrow_a \lambda z.\, \lambda t.\, zt$$

Aleshores l'exemple del TWICE el podríem escriure:

$$\begin{aligned}
\text{TWICE TWICE} &= (\lambda f.\, \lambda x.\, f(fx))\text{TWICE} \\
&\longrightarrow_\beta (\lambda x.\, \text{TWICE}(\text{TWICE}\ x)) \\
&= (\lambda x.\, \text{TWICE}(\lambda f.\, \lambda x.\, f(fx))x) \\
&\longrightarrow_a (\lambda x.\, \text{TWICE}((\lambda f.\, \lambda y.\, f(fy))x) \\
&\longrightarrow_b (\lambda x.\, \text{TWICE}((\lambda y.\, x(xy))
\end{aligned}$$

Ordres de reducció

Donada una λ-expressió, pot haver més d'un lloc on es pot aplicar β-reducció, per exemple:

$$(1)\ \ (\lambda x.\, x((\lambda z.\, zz)x))t \longrightarrow t((\lambda z.\, zz)t) \longrightarrow t(tt)$$

però també:

$$(2)\ \ (\lambda x.\, x((\lambda z.\, zz)x))t \longrightarrow (\lambda x.\, x(xx))t \longrightarrow t(tt)$$

Hi ha dues formes estàndard d'avaluar una λ-expressió:

- Avaluació en **ordre normal**: s'aplica l'estratègia **left-most outer-most**: Reduir la λ sintàcticament més a l'esquerra (1).
- Avaluació en **ordre aplicatiu**: s'aplica l'estratègia **left-most inner-most**: Reduir la λ més a l'esquerra de les que són més endins (2).

Ordres de reducció

En principi, podríem pensar que no importa l'ordre d'avaluació que utilitzem, perquè la β-reducció és **confluent**:

Si $t \to \cdots \to t_1$ i $t \to \cdots \to t_2$ llavors
$t_1 \to \cdots \to t_3$ i $t_2 \to \cdots \to t_3$

Tanmateix, si una expressió té una forma normal, aleshores la reducció en ordre normal la trobarà, però no necessàriament la reducció en ordre aplicatiu.

Per exemple, en ordre normal tenim:

$$(\lambda x.\, a)((\lambda y.\, yy)(\lambda z.\, zz)) \longrightarrow a$$

però en ordre aplicatiu:

$$(\lambda x.\, a)((\lambda y.\, yy)(\lambda z.\, zz)) \longrightarrow (\lambda x.\, a)((\lambda z.\, zz)(\lambda z.\, zz)) \longrightarrow \ldots$$

Macros

En el λ-càlcul, les funcions no reben noms.

Per facilitar-ne la escriptura, utilitzarem **macros** que representen funcions i les expandirem quan calgui, com vam fer a les transparències anteriors amb TWICE.

Les macros també es diuen **combinadors**.

⇒ És un recurs "meta" que no forma part del llenguatge (preprocessador).

Exemple:

$$\mathrm{ID} \equiv \lambda x.\, x$$

Llavors:

$$\begin{aligned} \mathrm{ID}\ \mathrm{ID} &\equiv (\lambda x.\, x)(\lambda x.\, x) \\ &\equiv (\lambda z.\, z)(\lambda x.\, x) \\ &\equiv \lambda x.\, x \\ &\equiv \mathrm{ID} \end{aligned}$$

Calculadores

Existeixen moltes calculadores de λ-càlcul *online*:

- https://www.cl.cam.ac.uk/~rmk35/lambda_calculus/lambda_calculus.html
- https://jacksongl.github.io/files/demo/lambda/index.htm
- http://www-cs-students.stanford.edu/~blynn/lambda/ (amb notació Haskell)

Naturals en λ-càlcul: Codificació

Podem definir els naturals en λ-càlcul d'aquesta manera:

$$\begin{aligned} 0 &\equiv \lambda sz.\, z \\ 1 &\equiv \lambda sz.\, s(z) \\ 2 &\equiv \lambda sz.\, s(s(z)) \\ 3 &\equiv \lambda sz.\, s(s(s(z))) \\ &\dots \\ n &\equiv \lambda sz.\, s^n z \end{aligned}$$

En altres paraules, el natural n és l'aplicació d'n cops la funció s a z.

Naturals en λ-càlcul: Codificació

Una codificació estranya? No tant:

Dec	Bin	Romà	Xinès	Devanagari
0	0		零	०
1	1	I	一	१
2	10	II	二	२
3	11	III	三	३
4	100	IV	四	४
⋮				⋮

L'important no és com es representen els naturals, sinó establir una bijecció entre la seva representació i $\mathbb{N}$.

Tampoc estem considerant-ne l'eficiència.

Naturals en λ-càlcul: Funció successor

La funció successor pot donar-se així:

$$\text{SUCC} \equiv \lambda abc.\, b(abc)$$

Apliquem-la a zero:

$$\begin{aligned}
\text{SUCC } 0 &\equiv (\lambda abc.\, b(abc))(\lambda sz.\, z) && \text{remplaçament macros}\\
&\equiv \lambda bc.\, b((\lambda sz.\, z)bc)) && \text{aplicació}\\
&\equiv \lambda bc.\, b((\lambda z.\, z)c)) && \text{aplicació}\\
&\equiv \lambda bc.\, b(c) && \text{aplicació}\\
&\equiv \lambda sz.\, s(z) && \text{renonenament de variables}\\
&\equiv 1 && 😄
\end{aligned}$$

Apliquem-la a un:

$$\begin{aligned}
\text{SUCC (SUCC } 0) &\equiv (\lambda abc.\, b(abc))(\lambda sz.\, s(z)) \\
&\ldots && \text{exercici}\\
&\equiv \lambda sz.\, s(s(z)) \\
&\equiv 2 && 😄
\end{aligned}$$

Naturals en λ-càlcul: Funció suma

La funció de suma:

$$\text{SUMA}\ x\ y\ \equiv\ x+y\ \equiv\ x\ \text{SUCC}\ y$$

o, també:

$$x+y\ \equiv\ \lambda pqxy.\,(px(qxy))$$

Proveu de sumar 3 i 2 amb les calculadores *online*.

Exercici: Com fer el producte?

Lògica en λ-càlcul: Booleans

Podem definir els booleans en λ-càlcul d'aquesta manera:

$$\begin{aligned} \text{TRUE} &\equiv \lambda xy.\, x \\ \text{FALSE} &\equiv \lambda xy.\, y \qquad \text{(com el zero!)} \end{aligned}$$

i definir els operadors lògics així:

$$\begin{aligned} \text{NOT} &\equiv \lambda a.\, a(\lambda bc.\, c)(\lambda de.\, d) \\ \text{AND} &\equiv \lambda ab.\, ab(\lambda xy.\, y) \\ \text{OR} &\equiv \lambda ab.\, a.\, (\lambda xy.\, x)b \end{aligned}$$

Exercici: Feu a mà les taules de veritat de la NOT i comproveu que és correcta.

Exercici: Utilitzeu les calculadores *online* per fer les taules de veritat de les operacions AND i OR i comprovar que són correctes.

Exercici: Escriviu TRUE i FALSE en Haskell, utilitzant funcions d'ordre superior.

Recursivitat en λ-càlcul

Sembla que sense poder donar noms a les funcions, el λ-càlcul no pugui donar suport a la recursivitat... però sí que es pot:

S'utilitza el **combinador Y**, anomenat *combinador paradoxal* o *combinador de punt fixe*, amb la següent proprietat:

$$\mathrm{YR} \;\equiv\; \mathrm{R}(\mathrm{YR})$$

Concretament, Y es defineix així:

$$\mathrm{Y} \;\equiv\; \lambda y.\,(\lambda x.\,y(xx))(\lambda x.\,y(xx))$$

Com podem veure:

$$\begin{aligned}
\mathrm{Y\,R} \;&\equiv\; (\lambda y.\,(\lambda x.\,y(xx))(\lambda x.\,y(xx)))\mathrm{R} \\
&\equiv\; (\lambda x.\,\mathrm{R}(xx))(\lambda x.\,\mathrm{R}(xx)) \\
&\equiv\; \mathrm{R}((\lambda x.\,\mathrm{R}(xx))(\lambda x.\,\mathrm{R}(xx))) \\
&\equiv\; \mathrm{R}(\mathrm{YR}) && \text{(per la línia anterior)}
\end{aligned}$$

Recursivitat en λ-càlcul

El combinador Y ens permet definir la funció factorial.

Sigui H la funció següent:

$$\lambda f.\, \lambda n.\, \mathrm{IF}(n = 0)\ 1\ (n \times (f\ (n - 1)))$$

podem veure com Y H funciona com el factorial:

$$YH1 \longrightarrow H(YH)1 = \lambda f.\, \lambda n.\, \mathrm{IF}(n = 0)1(n \times (f(n-1)))(YH)1 \longrightarrow$$

$$\lambda n.\, IF(n = 0)1(n \times (YH(n-1)))1 \longrightarrow IF(1 = 0)1(1 \times (YH(1-1))) \longrightarrow$$

$$1 \times (YH(1-1))) \longrightarrow YH0 \longrightarrow H(YH)0 =$$

$$\lambda f.\, \lambda n.\, \mathrm{IF}(n = 0)1(n \times (f(n-1)))(YH)0 \longrightarrow$$

$$\lambda n.\, \mathrm{IF}(n = 0)1(n \times (YH(n-1)))0 \longrightarrow \mathrm{IF}(0 = 0)1(0 \times (YH(0-1))) \longrightarrow 1$$

Universalitat del λ-càlcul

A partir d'aquí, ja només queda anar continuant fent definicions i anar-les combinant:

- ISZERO
- IF THEN ELSE
- …

Eventualment, es pot arribar a veure que qualsevol algorisme és implementable en λ-càlcul perquè pot simular a una màquina de Turing.

Teorema [Kleene i Rosser, 1936]: Totes les funcions recursives poden ser representades en λ-càlcul ($\Longleftrightarrow$ Turing complet).

A diferència de les màquines de Turing que són un model matemàtic d'una màquina *hardware* imperativa, el λ-càlcul només utilitza reescriptura i és un model matemàtic més *software* i funcional.