Teoria de la Programació Funcional

Paradigmes de Programació (04-05)

Mateu Villaret

Esquema del curs (i)

- Programació Funcional (motivació)
- La noció de funció
- El llenguatge λ-càlcul
  - λ-expressions
  - Notació
  - Variables: lliures vs. lligades
  - Substitució
  - Regles de transformació: α, β i η. (Càlcul)
  - Generalització i formes normals
  - λ-igualtat
  - estratègies de reducció

Esquema del curs (ii)

- λ-càlcul com a llenguatge de programació
  - Representacions elementals: booleans, condicional, naturals, ...
  - Operadors del punt fix
  - Recursivitat
  - El Factorial :-)
- Tipus (deducció) ???
- Verificació ???

Funcions

- **Abstracció**. Definir la funció: quins paràmetres té i que fa amb ells:

$$\overbrace{\lambda x.\, \lambda y.}^{parametres} \quad \overbrace{x + y}^{cos}$$

- **Aplicació**. Donada una funció i uns arguments, substituir els arguments pels paràmetres formals al cos de la definició:

$$(\lambda x.\, (\lambda y.\, x + y)) \quad 1 \quad 2$$

$$\Downarrow \text{subst.}$$

$$(\lambda y.\, 1 + y) \quad 2$$

$$\Downarrow \text{subst.}$$

$$1 + 2$$

λ-Terms

Suposem donat un conjunt enumerable de variables:
$\mathcal{X} = \{x, y, z, \ldots\}$.
Un λ-terme (només) pot ser:

$$
\begin{array}{rcll}
\Lambda & = & v & \text{on } v \in \mathcal{X} \\
 & | & (MN) & \text{on } M \in \Lambda \text{ i } N \in \Lambda \quad \textbf{Aplicació} \\
 & | & (\lambda v.\ M) & \text{on } v \in \mathcal{X} \text{ i } M \in \Lambda \quad \textbf{Abstracció}
\end{array}
$$

Exemples:

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

Notació (i)

- L'aplicació és *associativa cap a l'esquerra*:
  $M_1\, M_2\, M_3\, \ldots\, M_n$ significa $((\ldots((M_1\, M_2)\, M_3)\ldots)\, M_n)$.
- *L'àmbit* de λv arriba tant a la dreta com sigui possible:
  $\lambda v.\ M_1\, M_2 \ldots M_n$ significa $(\lambda v.\ (M_1\, M_2 \ldots M_n))$.
- Ens podem evitar λ's:
  $\lambda\, v_1\, v_2 \ldots v_n.\, M$ significa $(\lambda v_1.\ (\lambda v_2.\ (\ldots.(\lambda v_n.\ M)\ldots)))$.

  Exemples:

$$
\begin{array}{ll}
(\lambda x.\ x) & \Rightarrow \lambda x.\ x \\
((\lambda x.\ y)\ ((\lambda y.\ x)\ z)) & \Rightarrow (\lambda x.\ y)\ ((\lambda y.\ x)\ z) \\
((\lambda x.\ (\lambda y.\ ((x\ y)\ x)))\ (\lambda x.\ (y\ x))) & \Rightarrow \ldots
\end{array}
$$

Variable: lliures vs. lligades (i)

Les variables no són lliure o lligades, apareixen (ocorren) lliures o lligades.

- Una aparició d'una variable v en un λ-terme M és **lliure** si no està *dins de l'àmbit* d'una λv.
- Si una variable v apareix dins l'àmbit d'una λ-abstracció: $\lambda v.$ direm que ocorre **lligada**.

Per exemple a:

$$(\lambda y.\ z)(\lambda x.\ \mathbf{y}\ x)(\lambda y.\ \mathbf{x}\ y)$$

Podem moure parèntesis i modificar la llibertat de les ocurrències de les variables???

Variable: lliures vs. lligades (ii)

Definim $FV(\ )$ (free variables):

$$\begin{array}{lcll} FV(v) & = & \{v\} & \text{si } v \in \mathcal{X} \\ FV(M\ N) & = & FV(M) \cup FV(N) & \text{si } M \in \Lambda \text{ i } N \in \Lambda \\ FV(\lambda v.\ M) & = & FV(M) \setminus \{v\} & \text{si } v \in \mathcal{X} \text{ i } M \in \Lambda \end{array}$$

definim també $BV(\ )$ (bound variables):

$$\begin{array}{lcll} BV(v) & = & \emptyset & \text{si } v \in \mathcal{X} \\ BV(M\ N) & = & BV(M) \cup BV(N) & \text{si } M \in \Lambda \text{ i } N \in \Lambda \\ BV(\lambda v.\ M) & = & BV(M) \cup \{v\} & \text{si } v \in \mathcal{X} \text{ i } M \in \Lambda \end{array}$$

Els termes C tals que $FV(C) = \emptyset$ en direm **combinadors**

Substitució vs. reemplaçament

En "aplicar" una funció a un argument substituïm el paràmetre formal pel real:

$$(\lambda x.\ x + 1)\ 3 \Rightarrow 3 + 1$$

En presència de variables això és perillós:

$$(\lambda x, y.\ x + y)\ y \Rightarrow^{?} \lambda y.\ y + y$$

o bé:

$$(\lambda y.\ (\lambda x.\ y))\ x \Rightarrow^{?} \lambda x.\ x$$

Hem de fer una substitució que eviti la *captura de variables*.

Substitució (i)

La **substitució** la representarem com a $[v \mapsto M]$ (també es pot expressar com: $[M/v]$).
La substitució l'**aplicarem** a un terme:

M		$M[v \mapsto M']$
v		M'
v'	amb $v \neq v'$	v'
$M_1\ M_2$		$M_1[v \mapsto M']\ M_2[v \mapsto M']$
$\lambda v.\, M_1$		$\lambda v.\, M_1$
$\lambda v'.\, M_1$	on $v \neq v'$ i $v' \notin FV(M')$	$\lambda v'.\, M_1[v \mapsto M']$
$\lambda v'.\, M_1$	on $v \neq v'$ i $v' \in FV(M')$	$\lambda v''.\, M_1[v' \mapsto v''][v \mapsto M']$ on $v'' \notin FV(M')$ ni $v'' \notin FV(M_1)$

Substitució (ii)

Així:

$$(\lambda y.\, y\ x)[x \mapsto y]$$

quedaria:

$$(\lambda y.\, y\ x)[x \mapsto y] \equiv \lambda z.\, (y\ x)[y \mapsto z][x \mapsto y] \equiv \lambda z.\, (z\ x)[x \mapsto y] \equiv \lambda z.$$

i els següents:

- $(\lambda y.\, x(\lambda x.\, x))[x \mapsto (\lambda y.\, yx)]$
- $(y(\lambda z.\, xz))[x \mapsto (\lambda y.\, zy)]$

La Teoria del λ-càlcul (i)

Transformació, reducció, equivalència... α, β, η.

α-conversió
Tota abstracció de la forma $\lambda v.\, M$ es pot transformar en $\lambda v'.\, (M[v \mapsto v'])$ (sense capturar: v' és una *fresh variable*).
Per exemple:

$$\lambda x.\, x \longrightarrow_\alpha \lambda y.\, y$$

$$\lambda x.\, f x \longrightarrow_\alpha \lambda y.\, f y$$

però no:

$$\lambda x.\, \lambda y.\, f\ x\ y \longrightarrow_\alpha \lambda y.\, \lambda y.\, f\ y\ y$$

α-equivalència. Els λ-termes que es poden α-convertir es consideraran la mateixa entitat.

La Teoria del λ-càlcul (ii)

β-conversió

Tota aplicació de la forma $(\lambda v.\, M)\ N$ es pot transformar en $M[v \mapsto N]$.

Per exemple:

$$(\lambda x.\, f x)\ y \longrightarrow_\beta f y$$

$$(\lambda x.\, (\lambda y.\, f x y))\ a \longrightarrow_\beta \lambda y.\, f a y$$

però no:

$$(\lambda x.\, (\lambda y.\, f\ x\ y))\ (g y) \longrightarrow_\beta \lambda y.\, f\ (g y)\ y$$

Compte al identificar els redexs:

$$(\lambda x.\, \lambda y.\, f\ x\ y)\, a\, b$$

La Teoria del λ-càlcul (iii)

η-conversió

Tota abstracció de la forma $\lambda v.\,(M\,v)$ on v no apareix lliure a M, es pot transformar en M. Si tinguéssim la funció sin, llavors $\lambda x.\,(sin\ x)$ i sin denotarien la mateixa funció.
Per exemple:

$$\lambda x.\, f\,x \longrightarrow_\eta f$$

$$\lambda y.\, f\,x\,y \longrightarrow_\eta f\,x$$

però no:

$$\lambda x.\, f\,x\,x \longrightarrow_\eta f\,x$$

perquè x apareix lliure a $f\,x$.

La Teoria del λ-càlcul (iv)

Els Axiomes del λ-càlcul:

1. $\dfrac{s \longrightarrow_\alpha t \text{ o bé } s \longrightarrow_\beta t \text{ o bé } s \longrightarrow_\eta t}{s = t}$
2. $\dfrac{}{t = t}$
3. $\dfrac{s = t}{t = s}$
4. $\dfrac{s = t \text{ i } t = u}{s = u}$
5. $\dfrac{s = t}{su = tu}$
6. $\dfrac{s = t}{us = ut}$
7. $\dfrac{s = t}{\lambda x.\, s = \lambda x.\, t}$

La Teoria del λ-càlcul (v)

Els *mínims subtermes* als que es pugui aplicar una reducció els direm **redex**.

- $E_1 \longrightarrow_{\alpha(\beta\ o\ \eta)} E_2$ si E_2 es pot obtenir de E_1 $\alpha(\beta\ o\ \eta)$-convertint-ne algun *subterme*

Per exemple:

$$((\lambda x.\, \lambda y.\, f\, x\, y)\, a)\, b \longrightarrow_\beta (\lambda y.\, f\, a\, y)\, b \longrightarrow_\beta f\, a\, b$$

Direm que un terme està en X**-normal form** si no te cap X-redex.

Estratègies de Reducció (i)

Podem considerar que calcular serà trobar la $\beta\eta$-normal form d'un λ-terme...
Com decidir quins passos de reducció s'han de fer quan tenim més d'un redex???
Per exemple:

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

- innermost ($\approx$ greedy)
- outermost ($\approx$ lazy)

Estratègies de Reducció (ii)

Teorema 1 (Church-Rosser) *Si* $E_1 = E_2$ *llavors existeix un* E *tal que* $E_1 \rightarrow_{\beta(\ i\ \beta\eta)} E$ *i* $E_2 \rightarrow_{\beta(\ i\ \beta\eta)} E$.

Corolari 1 *La* forma normal *d'un* λ*-terme, si existeix, és única llevat d'*α*-conversió.*

Conseqüència: el λ-càlcul no és trivial.

La **seqüència d'ordre de reducció normal** consisteix en reduir primer, el redex més extern de més a l'esquerra:

Teorema 2 (de normalització) *Si* E *te forma normal, llavors, aplicant la seqüència de reducció de l'ordre de reducció normal, l'acabarem trobant.*

Alguns Combinadors "famosos"

`I`	$\equiv$	$\lambda x.\ x$	`I M`	=	`M`
`K`	$\equiv$	$\lambda x, y.\ x$	`K M N`	=	`M`
`K'`	$\equiv$	$\lambda x, y.\ y$	`K' M N`	=	`N`
`S`	$\equiv$	$\lambda x, y, z.\ xz(yz)$	`S M N L`	=	`M L (N L)`

Exercicis:

1. Demostra que `S K K = I`
2. Escriu un combinador `L` tal que `L M N = M ( N M) N`
3. Escriu el combinador `G` *golafre* tal que `G x = G`
4. Escriu un combinador `G'` tal que `G' x = x G'`

λ-definibilitat

- "Un llenguatge de programació és tot llenguatge que ens permet fer programes"...
- *“Els procediments mecànics són aquells que es podrien dur a terme per algun “agent” suficientment intel·ligent però al que no li calgués saber res sobre el que està calculant"*. [Turing]
- La definició dels λ-termes i el procés d’avaluació β-reducció, juntament amb la nostra interpretació fan del λ-càlcul un llenguatge de programació.
- Per tant definim: `bool`, `int`, `if--then--else`, tuples, llistes... i recursivitat.

Meta llenguatge

Per evitar escriure massa, usarem definicions de funcions amb meta-llenguatge i les notarem així:

$$\boxed{\text{SIGUI } \langle \textbf{nomexpressió} \rangle = \langle \lambda\text{-expressió} \rangle}$$

Per exemple si definíssim:

$$\boxed{\text{SIGUI } \textbf{IDENTITAT} = \lambda x.\, x}$$

tindríem que el terme:

$$\textbf{IDENTITAT}\ a \equiv (\lambda x.\, x)a$$

Els booleans (i)

De fet, hi ha moltes maneres de representar aquests element, aquí ho farem d'una manera clàssica:

$$\boxed{\text{SIGUI } \textbf{true} = \lambda x.\, \lambda y.\, x}$$

$$\boxed{\text{SIGUI } \textbf{false} = \lambda x.\, \lambda y.\, y}$$

$$\boxed{\text{SIGUI } \textbf{not} = \lambda t.\, t \ \textbf{false} \ \ \textbf{true}}$$

El condicional, el definim no com una funció pròpiament sino com una aplicació:

$$\boxed{\text{SIGUI } (E \rightarrow E_1 | E_2) = E \ E_1 \ E_2}$$

o bé:

$$\boxed{\text{SIGUI } \textbf{IfThenElse} = \lambda x\, y\, z.\, x\, y\, z}$$

Els booleans (ii)

not true	$=$		
	$=$	$(\lambda t.t$ **false true** $)$ **true**	{ *segons la def. de* **not**}
	$=$	**true false true**	{ β-*reduint*}
	$=$	$(\lambda x.\lambda y.\ x)$ **false true**	{ *segons la def. de* **true**}
	$=$	$(\lambda y.$ **false** $)$ **true**	{ β-*reduint*}
	$=$	**false**	{ β-*reduint*}
not false	$=$		
	$=$	$(\lambda t.t$ **false true** $)$ **false**	{ *segons la def. de* **not**}
	$=$	**false false true**	{ β-*reduint*}
	$=$	$(\lambda x.\lambda y.\ y)$ **false true**	{ *segons la def. de* **false**}
	$=$	$(\lambda y.\ y)$ **true**	{ β-*reduint*}
	$=$	**true**	{ β-*reduint*}

Els booleans (iii)

$$
\begin{array}{lll}
(\textbf{true} \rightarrow E_1 \mid E_2) & = & \\
\hline
 & = & \textbf{true}\ E_1\ E_2 \qquad \{\ \textit{def. del condicional}\} \\
 & = & (\lambda x.\lambda y.\ x) E_1\ E_2 \qquad \{\ \textit{def. de } \textbf{true}\} \\
 & = & (\lambda y.\ E_1) E_2 \qquad \{\ \beta\textit{-reduint}\} \\
 & = & E_1 \qquad \{\ \beta\textit{-reduint}\}
\end{array}
$$

$$
\begin{array}{lll}
(\textbf{false} \rightarrow E_1 \mid E_2) & = & \\
\hline
 & = & \textbf{false}\ E_1\ E_2 \qquad \{\ \textit{def. del condicional}\} \\
 & = & (\lambda x.\lambda y.\ y) E_1\ E_2 \qquad \{\ \textit{def. de } \textbf{false}\} \\
 & = & (\lambda y.\ y) E_2 \qquad \{\ \beta\textit{-reduint}\} \\
 & = & E_2 \qquad \{\ \beta\textit{-reduint}\}
\end{array}
$$

Els booleans (iv)

A partir d'aquestes definicions, podríem fer per exemple, la del AND:

$$\boxed{\text{SIGUI } \textbf{and} = \lambda x.\lambda y.\ (x \rightarrow y \mid \textbf{false})}$$

Exercici 1 *Com faríem la del* **OR***? i la del* **XOR***? Com podríem comprovar que són correctes?*

Parells i tuples (llistes) (i)

Representarem els parells ordenats: (E_1, E_2) i les funcions d'accés al primer **fst** element i al segon **snd**, com segueix:

$$\boxed{\text{SIGUI } \textbf{fst} = \lambda x.\, x \textbf{ true}}$$

$$\boxed{\text{SIGUI } \textbf{snd} = \lambda x.\, x \textbf{ false}}$$

$$\boxed{\text{SIGUI } \textbf{(} E_1\textbf{,} E_2 \textbf{)} = \lambda p.\, p\ E_1\ E_2}$$

que és doncs:

$$\textbf{fst (}E_1\textbf{,} E_2\textbf{)}$$

?

Parells i tuples (llistes) (ii)

Les n-tuples, o les llistes les podem definir doncs:

SIGUI **(** E_1**,** $E_2 \ldots$**,** E_n **)** $=$ **(** E_1 **, (** E_2 **, (** $\ldots$ **, (**E_{n-1}**,**E_n **)) …** **))**

per tant, l'i-èssim element de la n-tupla E (suposem ara $i < n$) el definiríem accedint al primer element de l'i-èssim -1 parell aniuat a E:

SIGUI **iessim-E** $=$ **fst(snd (snd (…snd (** E **)…)))**

i l'n-èssim?

Naturals (i)

$$\boxed{\text{SIGUI } \mathbf{0} = \lambda f.\, \lambda x.\, x}$$

$$\boxed{\text{SIGUI } \mathbf{1} = \lambda f.\, \lambda x.\, f\ x}$$

$$\boxed{\text{SIGUI } \mathbf{2} = \lambda f.\, \lambda x.\, f\ (f\ x)}$$

...

$$\boxed{\text{SIGUI } \mathbf{n}\ = \lambda f.\, \lambda x.\, \overbrace{f\ (\ldots (f}^{n}\ x) \ldots) \equiv \lambda f.\, \lambda x.\, f^n x}$$

$$\boxed{\text{SIGUI } \mathbf{suc} = \lambda n.\, \lambda f.\, \lambda x.\, n\ f\ (f\ x)}$$

Naturals (ii)

La suma i el producte segueixen la mateixa idea:

$$\boxed{\text{SIGUI } \textbf{suma} = \lambda m.\, \lambda n.\, \lambda f.\, \lambda x.\, m\ f\ (n\ f\ x)}$$

$$\boxed{\text{SIGUI } \textbf{prod} = \lambda m.\, \lambda n.\, \lambda f.\, \lambda x.\, m\ (n\ f)\ x}$$

La definició de la funció de testeig de zero tampoc és gaire complicada:

$$\boxed{\text{SIGUI } \textbf{eszero} = \lambda n.\, n\ (\lambda x.\ \ \textbf{false}\ )\ \textbf{true}}$$

Naturals (iii)

El que esperem de **prec** es que **prec** $0 = 0$ i **prec** $(n+1) = n$.
La idea clau es, donada $\lambda f x.\, f^n x$, eliminar una de les aplicacions de f.
El primer pas el farem a partir de la funció auxiliar **prefn** tal que aplicada sobre una funció qualsevol i una tupla que te com a primer element un booleà i com a segon un element qualsevol, es comporti com segueix:

$$\begin{aligned} \textbf{prefn} \quad f\ (\mathsf{true}, x) &= (\mathsf{false}, x) \\ \textbf{prefn} \quad f\ (\mathsf{false}, x) &= (\mathsf{false}, f\ x) \end{aligned}$$

La definició de **prefn** és:

SIGUI **prefn** $= \lambda f.\, \lambda p.$ **(false ,(fst** $p \rightarrow$ **snd** $p \mid f($ **snd** $p))$**)**

Naturals (iv)

de manera que:

$$\overbrace{\textbf{prefn}\ f(\textbf{prefn}\ f(\ \textbf{prefn}\ f(\cdots(\ \textbf{prefn}\ f}^{n+1}(\textbf{true}, x))\ldots))) = \textbf{(\ false\ ,}f^n x)$$

Ara ja podem definir la funció predecessor sobre els naturals:

$$\text{SIGUI}\ \textbf{prec} = \lambda n.\,\lambda f.\,\lambda x.\ \ \textbf{snd}(n(\textbf{prefn}\ \ f)\ \textbf{(\ true\ ,}\ x))$$

Exercitem...

1. Sumeu 2 i 1
2. quin es el predecessor de 3
3. com fariem la resta?
4. ...

Punt Fixe (i)

- Necessitem poder definir processos "iteratius"...
- Usarem recursivitat

$$\text{fact} = \lambda n.\ \textbf{(eszero}\ \ n \rightarrow 1 \mid *n\ (\text{fact}(\textbf{prec}\ \ n))\textbf{)}$$

- però no podem fer servir el propi nom!!!

Punt Fixe (ii)

Teorema 3 Del Punt Fixe

1. *Per a tot λ-terme F, existeix un X tal que $F\,X\;=\;X$.*
2. *Hi ha un combinador de punt fixe Y tal que per a qualsevol λ-terme F,*

$$F(Y\;F)\;=\;(Y\;F)$$

Demostració:

1.
2.

Punt Fixe (iii)

Definim un combinador de punt fixe:

$$\boxed{\text{SIGUI } \mathbf{Y} = \lambda f.\,(\lambda x.\,f(x\ x))\ (\lambda x.\,f(x\ x))}$$

és fàcil comprovar que realment **Y** és comporta com és d'esperar.

0	$\mathbf{Y}E$	$=$		
1		$=$	$\lambda f.\,(\lambda x.\,f(x\ x))\ (\lambda x.\,f(x\ x))\ E$	{ *segons la def. d*
2		$=$	$(\lambda x.\,E(x\ x))\ (\lambda x.\,E(x\ x))$	{ β-*redu*
3		$=$	$E\ ((\lambda x.\,E(x\ x))\ (\lambda x.\,E(x\ x)))$	{ β-*redu*
4		$=$	$E\ (\mathbf{Y}E)$	{ *per igualtat entre 0*

Punt Fixe (iv)

Ho és aquest?

$$\text{SIGUI } \mathbf{T} = (\lambda x.\,(\lambda y.\,y(x\ x\ y)))\ (\lambda x.\,(\lambda y.\,y(x\ x\ y)))$$

Punt Fixe (v)

Tornem a la definició (fraudulenta) del factorial:

$$\text{fact} = \lambda n.\ \textbf{(}\ \textbf{eszero}\ \ n \rightarrow 1 \mid *n\ (\text{fact}(\textbf{prec}\ \ n))\textbf{)}$$

que es pot escriure com a:

$$\text{fact} = (\lambda f.\, \lambda n.\ \textbf{(}\ \textbf{eszero}\ \ n \rightarrow 1 \mid *n(f\ (\textbf{prec}\ \ n))\textbf{)}\textbf{)}\ \text{fact}$$

Si ens fixem bé, és com si **fact** fos el punt fixe d'una funció (de l'expressió en λ-càlcul que li estem aplicant), així que podem definir **fact** com segueix:

$$\boxed{\text{SIGUI}\ \textbf{fact} = \textbf{T}\ (\lambda f.\, \lambda n.\ \textbf{(}\ \textbf{eszero}\ \ n \rightarrow 1 \mid *n(f\ (\textbf{prec}\ \ n))\textbf{)}\textbf{)}}$$

Punt Fixe (vi)

$$
\begin{array}{ll}
0 & \textbf{fact 2} \; = \\
1 & = \textbf{T} \overbrace{(\lambda f.\lambda n.\; \textbf{(}\; \textbf{eszero}\;\; n \rightarrow \textbf{1} \mid *n(f\;\textbf{(prec}\;\; n))\textbf{))}}^{\textbf{F}}\;\; \textbf{2} \\
2 & = (\lambda x.\,(\lambda y.\,y(x\;x\;y)))\;(\lambda x.\,(\lambda y.\,y(x\;x\;y)))\;\textbf{F}\;\;\textbf{2} \\
2 & = (\lambda y.\;y((\lambda x.\;(\lambda y.\;y(x\;x\;y)))\;(\lambda x.\;(\lambda y.\;y(x\;x\;y)))\;y))\;\textbf{F}\;\;\textbf{2} \\
3 & = \textbf{F}\; \overbrace{((\lambda x.\;(\lambda y.\;y(x\;x\;y)))\;(\lambda x.\;(\lambda y.\;y(x\;x\;y)))\;\textbf{F})}^{\textbf{fact}}\;\; \textbf{2} \\
4 & = (\lambda f.\lambda n.\; \textbf{(}\; \textbf{eszero}\;\; n \rightarrow \textbf{1} \mid *n(f\;\textbf{(prec}\;\; n))\textbf{))}\;\; \textbf{fact}\;\; \textbf{2} \\
5 & = \lambda n.\; \textbf{(}\; \textbf{eszero}\;\; n \rightarrow \textbf{1} \mid *n(\textbf{fact}\;\textbf{(prec}\;\; n))\textbf{)}\;\; \textbf{2} \\
6 & = \;\textbf{(}\; \textbf{eszero}\;\; \textbf{2} \rightarrow \textbf{1} \mid *\textbf{2}\;\textbf{(fact}\;\textbf{(prec}\;\; \textbf{2}))\textbf{)} \\
7 & = \textbf{*}\;\textbf{2}\;\;\textbf{(fact}\;\textbf{(prec}\;\; \textbf{2}\textbf{))} \\
8 & = \textbf{*}\;\textbf{2}\;\;\textbf{(fact}\;\textbf{1}\textbf{)} \\
9 & = \textbf{*}\;\textbf{2}\;\;\textbf{(T}\;(\lambda f.\lambda n.\; \textbf{(}\; \textbf{eszero}\;\; n \rightarrow \textbf{1} \mid *n\,(f\;\textbf{(prec}\;\; n))\textbf{))}\;\; \textbf{1}\textbf{)} \\
10 & = \ldots
\end{array}
$$

λ-càlcul tipat a la Curry (i)

L'associació de tipus als λ-termes fou originada per Haskell B. Curry (1934) i Alonzo Church (1940). Seguirem l'estil Curry.

Considerem un conjunt de tipus bàsics $\mathcal{B}$, el conjunt de variables de tipus $\mathcal{V}$, llavors els conjunt de tipus $\mathcal{T}$ és:

$$\mathcal{T} ::= \mathcal{B} \mid \mathcal{V} \mid \mathcal{T} \rightarrow \mathcal{T}$$

El constructor de tipus $\rightarrow$ és associatiu per la dreta. L'expressió $\tau_1 \rightarrow \tau_2$ denota el tipus de les funcions que van d'elements de tipus τ_1 a element de tipus τ_2.

λ-càlcul tipat a la Curry (ii)

El sistema de tipus a la Curry és pot definir com un sistema lògic amb

- un únic predicat que és l'assignació de tipus i que denotarem $M : \tau$ i que significa que el λ-terme M te tipus τ.
- A partir d'aquest predicat es pot definir el que direm una base o contexte que consistirà en un conjunt d'assignacions de tipus, sense contradiccions... És a dir, si $M : \tau_1 \in \Gamma$ i $M : \tau_2 \in \Gamma$ llavors $\tau_1 = \tau_2$.

λ-càlcul tipat a la Curry (iii)

Les derivacions dels tipus es construiran a partir d'assumpcions com $x : \tau$ amb les dues regles següents:

$$\frac{M : \tau_1 \to \tau_2 \qquad N : \tau_1}{MN : \tau_2}$$

$$\frac{\begin{array}{c}\boxed{x : \tau_1} \\ \vdots \\ M : \tau_2\end{array}}{\lambda x.M : \tau_1 \to \tau_2}$$

Si som capaços de derivar una assignació de tipus $M : \tau$ amb les assignacions de tipus que no hem enmarcat Γ, ho notarem $\Gamma \vdash M : \tau$.

λ-càlcul tipat a la Curry (iv)

Per exemple: sigui $\tau \in \mathcal{T}$ llavors de la següent derivació:

$$\dfrac{\dfrac{\boxed{f : \tau \to \tau} \quad \dfrac{\boxed{f : \tau \to \tau} \qquad \boxed{x : \tau}}{fx : \tau}}{f(fx) : \tau}}{\dfrac{\lambda x.f(fx) : \tau \to \tau}{\lambda f, x.f(fx) : (\tau \to \tau) \to \tau \to \tau}}$$

podem concloure

$$\vdash \lambda f, x.f(fx) : (\tau \to \tau) \to \tau \to \tau$$

λ-càlcul tipat a la Curry (v)

Propietats:

1. Si $M \rightarrow_\beta M'$ llavors:

$$\Gamma \vdash M : \tau \Rightarrow \Gamma \vdash M' : \tau$$

   al revés no!

2. Qualsevol terme al qual puguem assignar un tipus (és a dir, que sigui tipificable) és fortament normalitzant.
3. Qualsevol subterme d'un terme tipificable és tipificable.
4. Les variables de tipus són substituïbles per qualsevol tipus en els contextes, donant lloc a noves possibles derivacions.

λ-càlcul tipat a la Curry (vi)

Les següents preguntes són decidibles en el λ-càlcul amb tipus a la Curry que estem presentant:

1. Comprovació de tipus: donat un terme M i un tipus τ, podem derivar $\vdash M : \tau$?
2. Tipificabilitat: donat un terme M, existeix un τ tal que $\vdash M : \tau$?
3. Habitabilitat: donat un tipus τ, existeix un terme M tal que $\vdash M : \tau$?

Perdem la recursivitat...

Extensions com el λ-càlcul polimòrfic la recuperen però, la decibilitat de les preguntes sobre tipus...

Tipus en els LPs

- Un tipus es una col·lecció computable de valors que comparteixen algunes propietats estructurals: `Int, (Int,Int), [Char->Char], ...`
- Què és i què no és un tipus depèn del llenguatge en concret que considerem. (per exemple `['a',3,[]]` ...
- Utilitats dels tipus:
  - Organització i documentació del programa
  - Identificació i prevenció d'errors
  - ajut a l'optimització

Errors de tipus

- Un error de tipus es produeix quan a una expressió no se li pot associar cap tipus.
- També es produeixen quan es pretén manipular les expressions de manera que no es respectin les propietats estructurals del tipus que tenen.
- Els podem classificar en
  - Error hardware, considerem en C,
    `int x=3000; x(3); ...`
  - Disconformitat de la semàntica pretesa pel programador amb la de l'execució. Considerem
    `3 + 4.5`

Estratègies de control de tipus (i)

- "Run-time type checking" El control del tipus es fa en temps d'execució, de manera que en l'exemple
  `int x=3000; x(3); ...`
  just abans de fer la crida a la funció `x(3)` es comprovaria que efectivament `x` tingués tipus de funció.
- "Compile-time type checking" El control del tipus es fa en temps de compilació. Així en l'exemple anterior es detectaria ja en temps de compilació que la crida `x(3);` no és segura; de fet es detectaria que `x(3)` no te cap tipus assignable...

Estratègies de control de tipus (ii)

- Run-time type checking alenteix (ML és de 2 a 4 vegades més ràpid que LISP) l'execució però permet més flexibilitat al programar... C, LISP,...
- Habitualment els compile-time checking són "conservatius"... però troben l'error abans d'executar.
- Compte, el conjunt de programes que donen "run-time" error no és decidible!!!

```
if {expressio-booleana-que-es-pot-penjar}
    then (expressio amb error de tipus)
    else (expressio amb error de tipus)
```

Tipus i seguretat

Classificació de seguretat:

- Insegurs: C, C++... càsting, aritmètica de punters,...
- Quasi segurs: PASCAL, ADA, ... Permet gestió explicita de memòria...
- Segurs: (dynamic typing) LISP, SMALLTALK,... (static typing) JAVA, Haskell, ML,...
  `car x` en LISP

Type checking vs type inference

- El type checking consisteix en comprobar que efectivament el tipus d'una expressió és compatible amb el tipus declarat de l'expressió.
- En la inferència de tipus es mira al codi i es dedueix el tipus sense tenir en compte declaracions de tipus.
- Una bona solució és la proposada per HASKELL que disposa d'inferència de tipus però que si el programador en proporciona un, passa a ser type checking, d'aquesta manera s'aprofita els beneficis de documentació de les declaracions de tipus.

Inferència de tipus (Haskell) (i)

Seguirem els següents passos:

- contruir el "parse tree" de l'expressió (funció)
- posar els tipus coneguts a les fulles
- etiquetar els nodes que no tinguin tipus amb una variable diferent per cada node llevat que siguin fulles de l'arbre corresponent a la mateixa variable lligada
- aplicar els "constraints" corresponents als nodes aplicació i als nodes abstracció
- resoldre el problema d'unificació

Inferència de tipus (Haskell)(ii)

- Aplicació
  si tenim un terme $e_1\, e_2$, el seu parse tree és l'arbre corresponent al terme $e = @(e_1, e_2)$ de manera que $e :: t, e_1 :: t_1, e_2 :: t_2$, llavors podem imposar el constraint $t_1 \doteq t_2 \mapsto t$.
- Abstracció
  si tenim un terme $\lambda e_1.\, e_2$, el seu parse tree és l'arbre corresponent al terme $e = \lambda(e_1, e_2)$ de manera que $e :: t, e_1 :: t_1, e_2 :: t_2$, llavors podem imposar el constraint $t \doteq t_1 \mapsto t_2$.

Veure capítol 6 de *"Concepts in Programming Languages"* JOHN C. MITCHELL *Cambridge Press*.
També podeu consultar les transpes del seu llibre a la seva pàgina web.