Introducció a la Programació Funcional

Mateu Villaret i Auselle

15 de març de 2005

Resum

La programació funcional és un estil de programació basat en l'ús de les funcions des de la seva perspectiva més matemàtica.

Aquests apunts pretenen ser un punt de partida en el coneixement dels fonaments d'aquest paradigma: des del model de còmput λ-*càlcul* fins a la transformació i verificació de programes escrits amb llenguatges purament funcionals.

Índex

1 Presentació dels apunts

Aquests apunts pretenen ser una guia bàsica per al seguiment de les classes de l'assignatura de Programació Declarativa a ETIG/ETIS de l'UdG. Primerament motivarem l'interès de la programació funcional tot comparant-la amb la programació imperativa, després descriurem el model de còmput que hi ha darrera, el λ-càlcul, en la seva versió més simple, sense tipus i finalment mostrarem que un llenguatge de programació funcional no és gaire més que λ-càlcul ensucrat.

Per descomptat, tingueu en compte si us plau, que aquests apunts estan *en desenvolupament...* Els comentaris sobre la correcció i la claretat dels apunts són benvinguts (i esperats).

Aquests apunts estan basats en el curs *Introduction to Functional Programming* de John Harrison a Cambridge University.

2 Introducció

2.1 Què és?

És un estil de programació que utilitza les funcions des del *punt de vista matemàtic* per a la confecció de programes:

- una funció no te **efectes laterals**
- quan a una funció se li passa un argument, et "retorna" un resultat i **sempre** que li passis un **mateix paràmetre**, et torna el **mateix resultat**.

Així, un programa funcional és una expressió formada per funcions definides al mateix programa. Per tant, l'execució d'un programa funcional és l'avaluació d'una expressió.

2.2 Programació funcional, de la instrucció i l'assignació, al càlcul d'expressions

- La programació en el *paradigma imperatiu* eatà basada en al noció d'*estat*, típicament caracteritzat per una colecció de variables amb els seus valors[1]. Per tant, podríem dir que un programa és una sèrie de modificacions d'un estat fins arribar a un estat desitjat:

$$S_{inicial} \equiv S_0 \rightarrow_{c_1} S_1 \rightarrow \cdots \rightarrow_{c_n} \rightarrow S_n \equiv S_{final}$$

[1] De fet, si ens referim a l'estat en un programa escrit p. ex. en PASCAL, l'estat també constaria de la posició d'execució

Per exemple, en un programa (ben fet) d'ordenació d'un vector, l'estat inicial $S_{inicial} = \{\mathsf{vector} = [2,4,1,3], ...\}$ i l'estat final $S_{final} = \{\mathsf{vector} = [1,2,3,4], ...\}$.

Típicament aquestes comandes c_i són assignacions que van modificant l'estat. L'ordre d'execució és crucial, evidentment no te perquè ser el mateix el programa que executa les comandes amb aquest ordre: $\mathsf{c_1}; \mathsf{c_2}; ...; \mathsf{c_n}$ que amb aquest altre: $\mathsf{c_n}; \mathsf{c_{n-1}}; ...; \mathsf{c_1}$.

- La programació en el *paradigma funcional* no te variables, per tant no hi ha la noció d'estat. Així no podrem parlar d'assignacions ni d'execucions de comandes consecutivament i en ordre prefixat ja que no te sentit. En lloc de seqüenciar i buclejar, tindrem funcions recursives.

  D'altra banda, la programació funcional ens permetrà tractar les funcions com a ciutadans de primera classe ja que que podran ser paràmetres i/o resultats d'altres funcions.

  Programar funcionalment sol consistir més en dir què és el problema que no pas en dir quins son els passos que s'han de seguir per a resoldre'l.

  Per exemple, fer el *quicksort* d'una llista amb *HASKELL* seria:

```
quicksort []     =     []
quicksort (x:xs) =     (quicksort[y| y<-xs, y <x ])
                       ++ [x] ++
                       (quicksort[z| z<-xs, z > x])
```

  que és pot interpretar fàcilment com:

  - el *quicksort* de la llista buida *és* la llista buida,
  - i el *quicksort* de la llista que comenc a per `x` i segueix amb `xs` *és* afegir per la esquerra, a la llista que conté només `x`, la llista resultant de fer el *quicksort* als que són més petits que `x` i per la dreta, la llista resultant de fer el *quicksort* als que són més grans que `x`.

2.3 Els *pros* de la Programació funcional

1. *Els programes funcionals són més fàcils de manipular matemàticament que els imperatius*

   La clau és la noció de **transparència referencial**: el valor d'una expressió només depèn dels valors de les seves sub-expressions. Dit d'una altra manera, en un entorn definit, una expressió sempre te el mateix valor, per tant podem canviar iguals per iguals.

Aquesta característica permet *manipular* millor els programes essent així més fàcil:

- la demostració d'equivalència entre dos programes
- la demostració d'adequacitat d'un programa a la seva especificació
- la transformació de programes (per ex. per a la millora de l'eficiència)
- la construcció de funcions a partir de transformacions *naturals* d'altres funcions (per ex: la inversa)

De fet, els llenguatges funcionals usen la mateixa noció de variable que la matemàtica, no pas com els programes imperatius.

Vegem amb un exemple amb *PASCAL* el possible efecte nociu del mal us de l'assignació.

```
PROGRAM efecte_nociu_assignacio;
VAR
     flag: Boolean;

FUNCTION f(n:Integer):Integer;
BEGIN
     flag:= NOT flag;
     IF flag THEN
          f:=n
     ELSE
          f:= 2*n
END;

BEGIN
     flag:=TRUE;
     WRITE(f(1));
     WRITE(f(1));
     ...
     IF ((f(1)+f(2))=((f(2)+f(1)) THEN
           WRITE (‘‘res’’)
     ELSE
           WRITE (‘‘la suma no es commutativa?’’)
END.
```

En aquest cas una epressió f(1) no val sempre el mateix, ni l'expressió f(1)+f(2) és igual a f(2)+f(1), perdent així la commutativitat de la suma.

2. *Els programes funcionals tenen potents mètodes d'abstracció*

   L'abstracció en la programació és un element important. En la programació estructurada (imperativa) ja hi trobem un primer pas quan utilitzem accions i funcions. Per exemple, si volem sumar un vector de n enters, enlloc de fer: suma:= v[1]; suma:= suma + v[2]; ...; suma:= suma +v[n] faríem una funció de l'estil:

```
FUNCTION sumavect(v:ARRAY [1..n] of Integer):Integer;
VAR
   i,suma:Integer;
BEGIN
   suma:=0;
   FOR i:=1 TO n DO
      suma:=suma+v[i];
   sumavect:=suma
END;
```

   D'altra banda, imaginem-nos que el que vulguem fer pugui ser tant sumar com multiplicar. En programació imperativa[2] hauríem de crear una altra funció multiplicavect.En la programació funcional disposem d'una potent eina que son les **funcions d'ordre superior**: que són les funcions que tenen com a arguments altres funcions. Així per a l'exemple que estem comentant, podríem utilitzar una funció que es diu foldr a la que li passes una llista de valors, una funció i un element neutre de la funció i et "plega" la llista utilitzant la funció que li has passat i al final, l'element neutre.

$$\text{foldr } (+)\ 0\ [v[1], ..., v[n]] == v[1]+(v[2]+(...+(v[n]+0))...)$$

   o bé:

$$\text{foldr } (*)\ 1\ [v[1], ..., v[n]] == v[1]*(v[2]*(...*(v[n]*1))...)$$

   A part del foldr llenguatges com el *HASKELL* disposen d'altres funcions d'ordre superior que corresponen als esquemes de computació més utilitzats, així com també permeten definir noves funcions d'ordre superior.

   D'aquest aspecte de la programació funcional, se'n deriven dues conseqüències:

   - La modularitat és més fàcil i més natural

[2] De fet hi ha llenguatges com el *C* que permeten el pas de funcions com a paràmetres.

- Els programes resultants són més concisos, per tant,
  - poden ser més fàcils d'entendre,
  - més ràpids d'escriure i
  - amb menys possibilitats d'errors

3. *La programació funcional, aporta noves aproximacions algorísmiques als problemes*

   En la programació funcional "pura" apareix un nou concepte d'avaluació (d'expressions): l'avaluació **lazy** o **mandrosa** en contra de l'avaluació **eager** o **ansiosa**. La idea d'aquesta avaluació és que les sub-expressions d'una expressió no tenen perquè ser *totalment* avaluats si no és necessari per a l'avaluació de l'expressió total. Dit d'una altra manera, l'avaluació d'una expressió és postposada fins que el valor d'aquella expressió es necessita en algun lloc del còmput del programa. Les expressions només s'avaluen quan *realment es necessiten*. Com podem veure aquesta estratègia d'avaluació està totalment oposada amb l'ansiosa que avalua les expressions tant aviat com pot. Els arguments de les funcions no s'avaluen fins que aquests valors són necessaris en el *cos* de la funció. Si un argument no és necessitat en certa crida, aquest no s'avaluarà. Això, per exemple, ens permet construir expressions sense límits i passar-les com a arguments de funcions que només n'utilitzaran un tros finit.

   Una de les conseqüències més importants és que podem definir funcions que retornin expressions sense preocupar-nos de com es processaran o construir funcions sense haver-nos de preocupar de quan gran seran els arguments. Això també és un pas capça facilitar la modularitat.

4. *Predisposa a la paral·lelització*

   Per a poder paral·lelitzar còmputs, és necessari poder descomposar el programa en diferents trossos que es puguin calcular en paral·lel, repartir-los entre els processadors, coordinar l'execució i tornar a ajuntar els resultats dels còmputs com es necessitin. El fet de tenir transparència referencial i de no tenir la noció d'estat, permet que aquests trossos es puguin identificar immediatament ja que no hi ha dependències de temps en l'avaluació de les sub-expressions[3].

   Per a dur a terme tot aquest procés en programes imperatius, el compilador ha de ser molt més potent, deixant-se possiblement, trossos a paral·lelitzar. De fet hi ha llenguatges imperatius que permeten explicitar al programador quins són els trossos de codi que es poden avaluar en paral·lel. No obstant, en el paradigma funcional, el programador no se n'hauria de preocupar en absolut.

[3]Evidentment ens referim a expressions tals que no siguin una sub-expressió de l'altre

5. *La programació funcional juga un paper molt important en diverses àrees de la infomàtica*

   - D'una banda tenim l'àmbit de la intel·ligència artificial on llenguatges com el *LISP*, l'*SCHEME* o l'*ML* han jugat, i juguen, un paper tant important.
   - D'altra banda, també juguen un paper cada vegada més important en el desenvolupament de prototipus d'aplicacions. En diferents experiment s'ha pogut constatar que desenvolupar ràpidament prototipus amb llenguatges funcionals és més ràpid i més fàcil d'entendre que no pas amb llenguatges imperatius.

6. *L'estudi de la programació funcional ens permetrà aprofundir en conceptes informàtics tant importants com*

   - Els models de còmput,
   - els sistemes de tipus,
   - la recursivitat,
   - la verificació i/o transformació de programes
   - ...

7. *L'estudi de la programació funcional ens permetrà exercitar d'una manera nova la ment*

   L'haver d'atacar problemes d'una manera diferent a la que estem acostumats, ens pot ajudar a comprendre més a fons què és programar i ens ha de servir per a programar *millor*.

2.4 Els *contres* de la programació funcional?

1. *Els llenguatges funcionals són joguines de programació ineficients*

   Certament, als inicis eren molt lents en el còmput i necessitaven molta memòria. La recerca ha fet que això avui en dia no sigui així. Tot i aquesta millora hi ha qui creu, que els compiladors mai podran retallar la distància d'eficiència en respecte els compiladors imperatius ja que generen codi per a màquines amb arquitectura von Neumann.

2. *Els llenguatges funcionals no s'utilitzen ni s'utilitzaran al "mon real"*

   Si bé encara és més o menys cert, com ja hem comentat, la importància que te en certs àmbits, fa que la seva utilització vagi en augment. De fet no només en àmbits com aquest sinó que també en àmbits de la indústria com per exemples *ERICSSON* que va desenvolupar el seu propi llenguatge funcional *ERLANG*. Fins i tot s'ha desenvolupat un compilador de *Haskell* a la plataforma **.NET** de *Microsoft*.

És cert però que encara s'ha d'aprofundir i millorar aspectes com l"'input/output", la resposta per aplicacions en temps real, la programació de bases de dades, etc...

3. *Programar amb llenguatges funcionals no és natural*

   Segurament al principi pot donar aquesta impressió, no obstant, tinguem en compte que la majoria de la gent ha començat a programar amb paradigmes imperatius. De fet també podem preguntar-nos, per exemple, **què te de natural haver d'estar pensant en les adreces de memòria que ocupen empleats, comandes, números, noms, etc?**

2.5 Entorns

Dintre de la programació funcional també hi ha classificacions:

- Llenguatges funcionals purs: *FP*,*HASKELL*, *MIRANDA*, *HOPE*,...
- Llenguatges funcionals híbrids: *LISP*, *SCHEME*, *SML*,...
- D'altres variants, integrant nocions de concurrència: *ERLANG* o de programació lògica: *TOY*, etc...

2.6 Procés històric

Uns quants dels passos importants cap a la programació funcional actual:

- Als anys 30, *Alonzo Church* defineix el **λ-calculus**, model de còmput en que es basa la programació funcional.
- Cap als 50, *John McCarthy* crea el **LISP**, "pare" dels llenguatges funcionals; un dels seus descendents més directes és **SCHEME**.
- Al 1978, *J. Backus*, creador dels llenguatges **FORTRAN** i **Algol**, aposta per la programació funcional com a solució de la crisi del software, en la lectura que va fer al rebre el premi *Turing*:" *Can Programming be liberated from the von Neumann style*". En aquest mateix article, defineix el llenguatge **FP**
- A mitjans dels 70, apareix **ML** (encara fa concessions a l'estil imperatiu). A mitjans dels 80, s'estandaritza fent l'**SML**, s'en millora molt l'eficiència.
- Simultàniament, a mitjans dels 80, *D. Turner* crea **Miranda**, on ja s'inclouen nocions tant importants com: lazyness, pattern-matching,... (és de pagament)

- Al 1987 degut a la proliferació de llenguatges funcionals, es decideix definir un standard: **HASKELL**.
- Al 1998 es crea una versió estable que és el **HASKELL98**.

3 λ-càlcul

En aquesta secció parlarem de la teoria que hi ha darrera de la programació funcional. L'objectiu és acabar veient que programar, per exemple amb HASKELL, no és més que *endolçar* aquest model de còmput i que de fet els mecanismes necessaris per programar amb llenguatges funcionals ja hi són en el λ-càlcul.

Com hem comentat, una de les idees claus és la transparència referencial, doncs bé, aquest concepte queda capturat per la *substitució* que és el pas fonamental de l'*aplicació*. Així també, podrem definir funcions mitjançant l'altre concepte clau que és l'*abstracció*.

3.1 Aproximació

El λ-càlcul ens permet definir funcions sense noms i calcular l'aplicació d'aquestes a altres expressions.

L'**abstracció** seria el pas corresponent a "crear" una funció especificant

- quins paràmetres tindrà i
- que farà amb ells

així, informalment[4], una funció que rebés un número i li sumés 1 seria:

$$\lambda x.x + 1$$

El perquè utilitzem la λ per a notar l'abstracció d'una funció amb la x, per exemple, com a paràmetre, es deu a raons històriques[5].

L'**aplicació** consistiria en: donada una funció i un argument, **substituir** el paràmetre per l'argument en el cos de la funció. Seguint doncs l'exemple anterior, sumar 1 a 0 seria:

$$\begin{array}{c} (\lambda x.x + 1) \quad 0 \\ \Downarrow_{\text{subst.}} \\ 0 + 1 \end{array}$$

Sense la definició del 0, ni de l'operador + ni de l'1, no seríem capaços d'"avançar" més en el càlcul. Com veurem però, serem capaços de codificar en termes de λ-càlcul, λ-expressions, els elements de còmput que ens calguin.

[4] Com veurem, ni el + ni l'1 no formen part (inicialment) del llenguatge

[5] En el "*Principia Mathematica* de *Whitehead i Russell, 1910*, es feia servir $t[\hat{x}]$ per a funcions amb x com a paràmetre. *Church* la va "reutilitzar" així: $\hat{x}.t[x]$, però, pel que es veu, el mecanògraf no podia posar el barret: $\hat{}$ sobre la x i el va posar al costat així: $\wedge x.t[x]$, i això va ser transcrit per un altra mecanògraf en $\lambda x.t[x]$.

3.2 λ-expressions

Només hi ha tres tipus de λ-expressions:

1. **Variables**: $x, y, z, ...$
2. **Aplicació de funcions**: si E_1 i E_2 són λ-expressions, llavors $(E_1 E_2)$ també ho és, i *significa* el resultat d'aplicar la funció, o operador, E_1 a l'operand E_2.
3. **Abstracció**: si V és una variable i E una λ-expressió, llavors $\lambda V.\, E$ és una abstracció amb la *variable lligada* V i el *cos* E. Així $\lambda V.\, E$ denota la funció que quan s'aplica a un argument qualsevol E' esdevé la λ-expressió $E[V \mapsto E']$, o sigui, el *cos* de la funció però on hi havia la *variable lligada* V ara hi ha E'.

Utilitzant BNF, la sintaxis de les λ-expressions seria:

$$\begin{array}{rcl} \langle \text{variable} \rangle & ::= & x \mid y \mid z \mid \ldots \\ \langle \lambda\text{-expressió} \rangle & ::= & \langle \text{variable} \rangle \\ & \mid & (\ \langle \lambda\text{-expressió} \rangle\ \langle \lambda\text{-expressió} \rangle\ ) \\ & \mid & (\ \lambda\ \langle \text{variable} \rangle\ .\ \langle \lambda\text{-expressió} \rangle\ ) \end{array}$$

Per exemple, la λ-expressio:

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

seria la funció identitat. En aplicar-la a qualsevol altra λ-expressio ens *torna* aquesta altra expressió. En particular, la identitat aplicada a la identitat, és la identitat:

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

3.3 Notació

Com hem pogut ja observar, quan ens vulguem referir a una λ-expressió qualsevol, utilitzarem la lletra majúscula E possiblement amb subíndexos o "primes". Semblantment quan ens vulguem referir a una variable qualsevol utilitzarem la V, també amb subíndexos o "primes".

Per tal d'estaviar-nos parèntesis, tindrem en compte les següents convencions:

1. L'aplicació de funcions és associativa per l'esquerra: $E_1\, E_2\, E_3\, \ldots\, E_n$ significa $((\ldots((E_1\, E_2)\, E_3)\ldots)\, E_n)$. Per exemple:

$$\begin{array}{lll} E_1 E_2 & \text{significa} & (E_1 E_2) \\ E_1 E_2 E_3 & \text{significa} & ((E_1 E_2) E_3) \end{array}$$

2. L'àmbit de λV arriba tant a la dreta com sigui possible: $\lambda V.\, E_1\, E_2 \ldots E_n$ significa $(\lambda V.\, (E_1\, E_2 \ldots E_n))$

3. $\lambda V_1\, V_2 \ldots V_n.\, E$ significa $(\lambda V_1.\, (\lambda V_2.\, (\ldots.(\lambda V_n.\, E)\ldots)))$. Per Exemple:

$$\begin{array}{lll} \lambda x\, y.\, E & \text{significa} & (\lambda x.\, (\lambda y.\, E)) \\ \lambda x\, y\, z.\, E & \text{significa} & (\lambda x.\, (\lambda y.\, (\lambda z.\, E))) \end{array}$$

3.4 Variables lliures i variables lligades

Una aparició d'una variable V en una λ-expressió és **lliure** si no està *dins de l'àmbit* d'una λV. Si una variable no és lliure, direm que és **lligada**. Per exemple a:

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

les aparicions de les variables en negreta són lliures, les altres són lligades.

Podem definir el conjunt de variables lliures FV (free variables) d'una λ-expressió recursivament com segueix:

$$\begin{array}{lcll} FV(V) & = & \{V\} & \text{si } V \text{ és una variable} \\ FV(E_1E_2) & = & FV(E_1) \cup FV(E_2) & \text{si } E_1 \text{ i } E_2 \text{ són dues } \lambda\text{-expressions} \\ FV(\lambda V.E) & = & FV(E) \setminus \{V\} & \text{si } V \text{ és una variable i } E \text{ una } \lambda\text{-expressió} \end{array}$$

també podem definir-ne el conjunt de variables lligades BV (bound variables):

$$\begin{array}{lcll} BV(V) & = & \emptyset & \text{si } V \text{ és una variable} \\ BV(E_1E_2) & = & BV(E_1) \cup BV(E_2) & \text{si } E_1 \text{ i } E_2 \text{ són dues } \lambda\text{-expressions} \\ BV(\lambda V.E) & = & BV(E) \cup \{V\} & \text{si } V \text{ és una variable i } E \text{ una } \lambda\text{-expressió} \end{array}$$

A partir d'aquestes dues nocions sobre variables, som capaços de definir *correctament* la substitució.

3.5 Substitució

Substitució significa canviar. En λ-càlcul substituirem variables per λ-expressions. La substitució la notarem així: $[V \mapsto E_2]$ i aplicada a un λ-terme E_1: $E_1[V \mapsto E_2]$ serà el λ-terme resultant de substituir tota aparició *lliure* de V a E_1 per E_2. Per tal de poder definir les *regles del càlcul correctament*, hem de definir una substitució [6] que **eviti la captura de variables**, per exemple:

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

no ens interessa que sigui:

$$\lambda x\, y.\, y$$

[6] Cal tenir en compte, que a la bibliografia, la substitució s'expressa de moltes maneres, entre elles, $[x \mapsto E]$ es pot expressar com: $[E/x]$

ja que com veiem la variable lliure de la substitució y, ha passat a ser lligada.

Definirem la substitució vàlida o correcta (que és la que usarem) amb la forma genèrica $E[V \mapsto E']$ recursivament en funció de com sigui E i de les variables involucrades, com segueix:

E		$E[V \mapsto E']$	
V		E'	
V'	amb $V \neq V'$	V'	
$E_1\ E_2$		$E_1[V \mapsto E']\ E_2[V \mapsto E']$	
$\lambda V.\, E_1$		$\lambda V.\, E_1$	
$\lambda V'.\, E_1$	on $V \neq V'$ i $V' \notin FV(E')$	$\lambda V'.\, E_1[V \mapsto E']$	
$\lambda V'.\, E_1$	on $V \neq V'$ i $V' \in FV(E')$	$\lambda V''.\, E_1[V' \mapsto V''][V \mapsto E']$	on $V'' \notin FV(E')$ ni $V'' \notin FV(E_1)$

Per a veure'n un exemple analitzarem $(\lambda y.\, y\ x)[x \mapsto y]$. Com que y és lliure a y, hem d'aplicar l'últim cas de la substitució. Agafem per exemple, z com a V'' ja que no apareix ni a $(y\ x)$ ni a y (de fet, amb que n'agafem una que no aparegui enlloc, en tenim prou) :

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

Exercici 1 *Feu les següents substitucions:*

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

3.6 Regles de transformació

Com veurem, el λ-càlcul ens permetrà representar objectes com números, booleans, cadenes, etc. Per exemple, una expressió aritmètica com $(2+3)*5$ es pot representar com una λ-expressió, i el seu "valor" també. De fet, la mateixa representació que escollim només serà tal, si és capaç d'emular el còmput dels objectes que representen: $2, 3, +, *$, i per tant, si de la λ-expressió $(2+3)*5$ en podem obtenir la λ-expressió que representa el valor de simplificar la suma i el producte, o sigui 25.

La transformació de les λ-expressions es durà a terme mitjançant unes regles molt generals, de manera que quan s'apliquin a λ-expressions que representin expressions arimètiques (correctament), estaran simulant l'avaluació de la suma, del producte, etc... i quan s'apliquin a λ-expressions que representin els booleans i les operacions booleanes, estaran simulant l'avalució de l' and, l'or, etc.

Hi ha tres tipus de λ-*transformacions*, o *conversions*, o *reduccions*[7]: α-conversió, β-conversió i η-conversió. Cal tenir en compte, que les substitu-

[7]De fet, les regles que donarem aquí, les donarem orientades i típicament ens hi referirem com a reduccions, però es poden estudiar com a equivalències en una teoria.

cions que hi aparèixen han de ser vàlides, és a dir, no han de "capturar" variables.

La més important per a nosaltres serà la β-conversió que ens permetrà simular diferents mecanismes d'avaluació. L'α-conversió tindrà més a veure amb la manipulació tècnica dels noms de les variables lligades, mentre que l'η-conversió te a veure amb l'extensionalitat.

En general, a les expressions a les que se'ls pugui aplicar les regles les anomenarem **redex**, de **red***ucible* **ex***pression*.

3.6.1 α-conversió

Tota abstracció de la forma $\lambda V.\, E$ es pot transformar en $\lambda V'.\, (E[V \mapsto V'])$. Quan una λ-expressió E_1 s'α-converteixi en E_2 ho escriurem $E_1 \longrightarrow_\alpha E_2$. Aquesta regla ens ve a dir que el "nom" de les variables lligades no és rellevant i que el podem canviar pel que vulguem sempre que fem una substitució vàlida. Per exemple:

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

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

però no:

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

ja que per a canviar la primera λ-abstracció de x a y hauríem de fer la substitució $\lambda y.\, ((\lambda y.\, f\ x\ y)[x \mapsto y])$ que en realitat ens faria canviar també el nom de la variable lligada y, donant-nos per exemple com a α-conversió correcta:

$$\begin{array}{l} \lambda y.\, ((\lambda y.\, f\ x\ y)[x \mapsto y]) \\ \quad \Downarrow \\ \lambda y.\, \lambda z.\, ((f\ x\ y)[y \mapsto z][x \mapsto y]) \\ \quad \Downarrow \\ \lambda y.\, \lambda z.\, f\ y\ z \end{array}$$

3.6.2 β-conversió

Tota aplicació de la forma $(\lambda V.\, E_1)\ E_2$ es pot transformar en $E_1[V \mapsto E_2]$. Quan una *lambda*-expressió E es β-converteixi en E' ho escriurem $E \longrightarrow_\beta E'$. Aquesta regla és com l'avaluació d'una crida a una funció en un llenguatge de programació: el cos E_1 de la funció $\lambda V.E_1$ s'avalua en un entorn on el "paràmetre formal" V s'ha lligat al "paràmetre real" E_2, és com si "gastéssim" la λ-abstracció. 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:

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

Pot costar una mica familiaritzar-se amb les convencions comentades abans per identificar correctament els "redex", per exemple:

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

correctament parentitzada seria:

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

que correspon a la forma de λ-expressió:

$$((\lambda x.\,E)a)b$$

on

$$E = (\lambda y.\,f\ x\ y)$$

Fixem-nos per tant, en que no tota la λ-expressió $(((\lambda x.\,(\lambda y.\,((f\ x)\ y)))\,a)\,b)$ és un β-redex, sinó que només una sub-expressió ho és.

3.6.3 η-conversió

Tota abstracció de la forma $\lambda V.\,(EV)$ on V no apareix lliure a E, es pot transformar en E. Quan E_1 s'η-converteixi en E_2 ho escriurem $E_1 \longrightarrow_\eta E_2$. Aquesta regla expressa la propietat que dues funcions són iguals si donen el mateix resultat sempre que s'apliquen als mateixos arguments. Si tinguéssim la funció sin, llavors $\lambda x.\,(sin\ x)$ i sin denotarien la mateixa funció. Per exemple:

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

$$\lambda y.\,fxy \longrightarrow_\eta fx$$

però no:

$$\lambda x.\,fxx \longrightarrow_\eta fx$$

perquè x apareix lliure a fx.

3.7 Transformació generalitzada, reducció i forma normal

Les definicions de $\longrightarrow_\alpha$, $\longrightarrow_\beta$ i $\longrightarrow_\eta$ es poden generalitzar com segueix:

- $E_1 \longrightarrow_\alpha E_2$ si E_2 es pot obtenir de E_1 α-convertint-ne algun *subterme*
- $E_1 \longrightarrow_\beta E_2$ si E_2 es pot obtenir de E_1 β-convertint-ne algun *subterme*
- $E_1 \longrightarrow_\eta E_2$ si E_2 es pot obtenir de E_1 η-convertint-ne algun *subterme*

Per exemple:

$$((\lambda x.\, \lambda y.\, fxy)a)b \longrightarrow_\beta (\lambda y.\, fay)b \longrightarrow_\beta fab$$

El primer pas és generalitzat ja que el terme sencer no és un β-redex, sinó que ho és el subterme: $(\lambda x.\, \lambda y.\, fxy)a$.

D'alguna manera, podem dir que **calcular amb aquest llenguatge consisteix en aplicar les regles de reducció fins que vegem que no en podem aplicar cap més** (a part, d'α-conversions és clar) arribant al que en diem la **forma normal**. Bàsicament, la regla que aplicarem serà la β-reducció que a més te una propietat molt important: si tenim diversos *β-redexes* tant se val en quin ordre els resolguem que sempre podrem acabar obtenint el mateix [8].

En general, per notar que un λ-terme s es **redueix** en un altre t, en un cert nombre de passos utilitzant les regles de transformació, farem servir: $s \longrightarrow t$.

3.8 λ-igualtat

A partir d'aquesta generalització, podríem definir la **λ-igualtat** mitjançant aquestes regles que volen dir: *si el que hi ha sobre la fracció és cert, llavors s'en pot deduïr el que és a sota*:

1. $$\frac{s \longrightarrow_\alpha t \text{ o bé } s \longrightarrow_\beta t \text{ o bé } s \longrightarrow_\eta t}{s = t}$$

2. reflexivitat:
$$\frac{}{t = t}$$

3. simetria:
$$\frac{s = t}{t = s}$$

4. transitivitat:
$$\frac{s = t \text{ i } t = u}{s = u}$$

5. $$\frac{s = t}{su = tu}$$

6. $$\frac{s = t}{us = ut}$$

[8] Com veurem, quan es tracti d'escollir una estratègia d'aplicació, n'hi haurà una que te garantida l'acabament, si és que aquest és possible, i una altra que no.

7.

$$\frac{s = t}{\lambda x.\, s = \lambda x.\, t}$$

Compte perquè una cosa és la igualtat sintàctica i l'altra la igualtat definida per aquesta teoria.

Exercici 2 *Utilitzant aquestes regles, seríem capaços de demostrar que:* $(\lambda x.\, \lambda y.\, y)\ a\ b = (\lambda z.\, z)\ b$ *o no?*

3.9 Estratègies de reducció

Com hem comentat, un programa funcional és una expressió i *executar-lo* significa *avaluar-la*. Per tant, pel que hem vist, el que hem de fer per a avaluar és anar aplicant reduccions fins que no puguem reduir res més, fins que estigui totalment avaluada. Com ho hem de fer però, per tal de saber **en cada moment quin *redex* hem de tractar?** De fet, hi ha λ-expressions que si les avalues no "acabes" mai, per tant, hi ha λ-expressions que depenent de quina estratègia de reducció facis servir, el procés de reducció (avaluació) pot acabar i pot no acabar. Per exemple, si sempre resolem el *redex més intern*:

$$\begin{array}{ll} & (\lambda x.\, y) \quad ((\lambda x.\, x\ x\ x)(\lambda x.\, x\ x\ x)) \\ \longrightarrow & (\lambda x.\, y) \quad ((\lambda x.\, x\ x\ x)(\lambda x.\, x\ x\ x)(\lambda x.\, x\ x\ x)) \\ \longrightarrow & (\lambda x.\, y) \quad ((\lambda x.\, x\ x\ x)(\lambda x.\, x\ x\ x)(\lambda x.\, x\ x\ x)(\lambda x.\, x\ x\ x)) \\ \longrightarrow & \ldots \end{array}$$

d'altra banda, si per la mateixa expressió reduïm el *més extern i de més a l'esquerra*:

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

Aquest exemple, en realitat, respon a uns teoremes molt importants sobre el λ-càlcul.

Teorema 1 (Church-Rosser) *Si* $E_1 =_\lambda E_2$ *llavors existeix un* E *tal que* $E_1 \to_\lambda E$ *i* $E_2 \to_\lambda E$.

Aquest teorema ens ve a dir que qualsevol λ-expressió es pot avaluar en qualsevol ordre. Suposem que un λ-terme E és avaluat de dues maneres diferents i arriba a dues formes normals diferents: E_1 i E_2. Com que aquestes venen de reduir E, podem dir que $E_1 =_\lambda E_2$, llavors, com que E_1 i E_2 estan en forma normal, l'única regla que es pot aplicar és l'α-conversió, així, aplicant al teorema arribem a tenir una única forma normal llevat d'α conversió.

Per tant, com a corol·lari podem afirmar que:

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

Una aplicació doncs d'aquests resultats serveix per a demostrar que la teoria del λ-càlcul no és trivial, és a dir, que no tots els termes són iguals ja que com podem veure, els termes:

$$\lambda f.\, \lambda x.\, fx \qquad \lambda f.\, \lambda x.\, ffx$$

estan en forma normal i no són iguals mòdul α-conversió.

Però, quina estratègia de reducció hem de seguir per arribar a la forma normal?

Hem de seguir la **seqüència de l'ordre de reducció normal**: reduir primer el redex més extern de més a l'esquerra. Aquest ordre te una propietat molt interessant:

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

Això també ens permet garantir que qualsevol seqüència de passos de reducció que acabi, donarà sempre el mateix resultat. Mai és tard per abandonar una estratègia i passar a l'ordre normal.

Evidentment, en termes de programació funcional, l'estratègia de reducció te a veure amb els conceptes d'avaluació *lazy* i d'avaluació *eager*...

4 λ-càlcul com a llenguatge de programació

El concepte de llenguatge de programació, es pot definir "fent trampa" mitjançant la següent autoreferència: *un llenguatge de programació, és tot aquell llenguatge que ens permet fer* **programes**.

La idea de la construcció de procediments mecànics per l'obtenció de resultats matemàtics, sempre ha estat considerada. Als anys 30 els matemàtics i lògics del moment, estaven treballant en la resolució del que es coneix com al "problema de la decisió": "Existeix algun procediment mecànic sistemàtic que ens permeti saber sempre si una fórmula lògica de primer ordre és vàlida?".

Per tant ara ens caldria definir que és un programa, o un **algorisme**, o un **procediment mecànic sistemàtic** (com es definia als orígens).

Turing en va fer una definició acurada: *"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"*. El mateix Alan Turing va idear un sistema formal que permetia expressar aquesta mena de procediments mecànics o "programes", les "Màquines de Turing". Aquest formalisme és el que hi ha darrera la programació imperativa que avui coneixem, ja que de fet ens permet representar la noció d'estat, la memòria i les modificacions de l'estat.

El λ-càlcul amb la β-reducció ens permet expressar també aquesta mena de procediments segons les condicions de Turing. És la pròpia definció dels termes i la interpretació que nosaltres en fem, que al ser avaluats ens calcularan el que volem. Aquesta de fet és la base de la programació funcional.

El propi Church va postular que en realitat, el conjunt de funcions que es poden expressar mitjançant termes λ-càlcul, formalitzen la idea intuitiva de procediment mecànic. Més tard, Turing va demostrar que el λ-càlcul i les màquines de Turing, podien calcular el mateix.

Anem a veure com amb λ-càlcul podem representar els elements típics dels llenguatges de programació que vulguem com: *booleans, enters, tuples, ...* amb les respectives operacions, expressió *if then else*, recursivitat etc.

4.1 Representacions bàsiques

Per tal de facilitar la lectura dels apunts i la comprensió de les expressions, utilitzarem un "endolçament" del λ-càlcul per mitjà de la definició de termes, donant-los-hi noms. Aquestes definicions que formen el meta-llenguatge dels apunts, les notarem així:

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

Per exemple si definíssim:

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

tindríem que el terme:

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

Com veurem, costa a priori, trobar el sentit a les definicions dels termes que proposem, però és quan els "utilitzem" que s'en veu bé el funcionament i la coherència.

4.1.1 Booleans i condicional

En aquesta secció, definirem les λ-expressions **true**, **false**, **not** i el condicional $(E \rightarrow E_1|E_2)$ que representaran respectivament, els valors de veritat CERT, FALS, NOT i el condicional: si E llavors E_1 sino E_2.

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

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

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

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

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

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

Si ho volguéssim representar com una funció, hauríem de considerar una funció que rep tres paràmetres, primer l'expressió booleana després l'expressió corresponent al cas cert i finalment la corresponent al cas fals:

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

Veiem com realment, la definició de **true**, **false** i **not** es comporten com és d'esperar:

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*}

així com el condicional, suposant que tenim dues λ-expressions qualsevols E_1 i E_2:

$(\mathbf{true} \rightarrow E_1 \mid E_2)$	$=$		
	$=$	$\mathbf{true}\ E_1\ E_2$	$\{$ *segons la def. del condicional* $\}$
	$=$	$(\lambda x.\lambda y.\ x)E_1\ E_2$	$\{$ *segons la def. de* **true** $\}$
	$=$	$(\lambda y.\ E_1)E_2$	$\{$ *β-reduint* $\}$
	$=$	E_1	$\{$ *β-reduint* $\}$
$(\mathbf{false} \rightarrow E_1 \mid E_2)$	$=$		
	$=$	$\mathbf{false}\ E_1\ E_2$	$\{$ *segons la def. del condicional* $\}$
	$=$	$(\lambda x.\lambda y.\ y)E_1\ E_2$	$\{$ *segons la def. de* **false** $\}$
	$=$	$(\lambda y.\ y)E_2$	$\{$ *β-reduint* $\}$
	$=$	E_2	$\{$ *β-reduint* $\}$

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

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

Exercici 3 *Com faríem la del* OR*? i la del* XOR*? Com podríem comprovar que són correctes?*

4.1.2 Parells i tuples

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

$$\boxed{\mathsf{SIGUI}\ \mathbf{fst} = \lambda x.\, x\ \mathbf{true}}$$

$$\boxed{\mathsf{SIGUI}\ \mathbf{snd} = \lambda x.\, x\ \mathbf{false}}$$

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

així suposant que tenim dues λ-expressions qualsevols E_1 i E_2:

$\mathbf{fst}\ (E_1, E_2)$	$=$		
	$=$	$(\lambda p.\, p\ \mathbf{true}\ )(E_1, E_2)$	$\{$ *segons la def. de* **fst** $\}$
	$=$	$(\lambda p.\, p\ E_1\ E_2)\mathbf{true}$	$\{$ *segons la def. de la tupla* $\}$
	$=$	$\mathbf{true}\ E_1\ E_2$	$\{$ *β-reduint* $\}$
	$=$	$(\lambda x.\, \lambda y.\, x)E_1\ E_2$	$\{$ *segons la def. de* **true** $\}$
	$=$	$(\lambda y.\, E_1)E_2$	$\{$ *β-reduint* $\}$
	$=$	E_1	$\{$ *β-reduint* $\}$

És fàcil veure com **snd** també funciona. A partir d'aquí es podria definir sense massa dificultat, les n-tuples com a tuples de tuples:

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

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:

$$\boxed{\text{SIGUI } \textbf{iessim-E } = \textbf{fst(snd (snd (...snd (} E \textbf{)...)))}}$$

Exercici 4 *Com accediríem a l'n-èssim?*

4.1.3 Nombres

Per a definir els nombres, l'objectiu serà "associar" a cada nombre **0**, **1**, ..., una λ-expressió, així com també definir les λ-expressions que representen les operacions aritmètiques bàsiques: **suc**, **suma**, **eszero**, ...

La representació que presentem es deu a Church i està inspirada en el sistema numèric unari (el dels palets). Els números seran funcions que tindran dues variables lligades f i x. Així cada f que aparegui al cos de la funció representarà una unitat: tantes fs tantes unitats, i podríem dir que la x seria el final del número.

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

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

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

...

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

La construcció de les operacions senzillament ha de donar-nos expressions d'acord amb la definició que hem dit, així la funció **suc** consistirà en afegir una f al número que li passin. Això ho aconseguim, aplicant el número a la variable lligada f que representarà les "noves" unitats i a $f\ x$ que representaria el zero, però si us fixeu, ara enlloc de ser només x és $f\ x$, una unitat més[9]:

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

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:

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

[9]Recordeu que el nom de les variables no és significatiu, $\lambda g.\, \lambda z.\, gz$ també serviria com a 1

L'operació del predecessor **prec** és una mica més complicada... El que esperem de **prec** es que **prec** $0 = 0$ i **prec** $(n+1) = n$. La idea clau va ser, 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:

$$\begin{array}{lcl} \textbf{prefn} \quad f\ (\text{true}, x) & = & (\text{false}, x) \\ \textbf{prefn} \quad f\ (\text{false}, x) & = & (\text{false}, f\ x) \end{array}$$

La definició de **prefn** és:

$$\boxed{\mathsf{SIGUI}\ \textbf{prefn} = \lambda f.\, \lambda p.\, (\textbf{false}, (\textbf{fst}\ \ p \rightarrow \textbf{snd}\ \ p \mid f(\ \textbf{snd}\ p)))}$$

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:

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

Exercici 5 *Comproveu que efectivament* **prec** $\lambda f.\, \lambda x.\, f(fx) = \lambda f.\, \lambda x.\, fx$

4.2 Definicions per recursió

A hores d'ara, ja hem vist com podem construir elements de tipus de dades simples com els *naturals*, o els *booleans* i d'altres més complexos com les *tuples* o les *n-tuples*. També hem vist com es poden realitzar operacions bàsiques amb ells: *and-lògic*, *suma*, *producte*, ... També hem vist com podíem fer el *condicional*. Ara, el que ens cal és trobar un mètode per a dur a terme l'estructura algorísmica bàsica, la **iteració**. En λ-càlcul veurem com definir funcions per **recursió**, que en certa manera podríem dir que és en la programació funcional el que fa de iteració.

A primera vista el problema és com referir-nos a la mateixa funció que estem definint, ja que en λ-càlcul no tenim noms de funció, recordem que aquests noms només formen part del "meta-llenguatge".

4.2.1 Els Operadors de Punt Fix

El punt fix d'una funció qualsevol f són aquells x tals que $f(x) = x$.[10]

Els **operadors de punt fix** són aquelles λ-expressions que aplicades a d'altres funcions ens en donen sempre un punt fixe, és a dir, si **Y** és una funció tal que aplicada a qualsevol altra funció f, satisfà que:

[10] Per exemple, un punt fix de la funció matemàtica *sinus* és el 0 ja que $sinus(0) = 0$.

$$\mathrm{f}\ (\ \mathbf{Y}\ \mathrm{f}\ ) = \mathbf{Y}\ \mathrm{f}$$

llavors direm que **Y** és un operador de punt fix.
En λ-càlcul hi ha infinits operadors de punt fix, definim-ne un:

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

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. de* $\mathbf{Y}\}$
2		$=$	$(\lambda x.\,E(x\ x))\ (\lambda x.\,E(x\ x))$	$\{$ *β-reduint*$\}$
3		$=$	$E\ ((\lambda x.\,E(x\ x))\ (\lambda x.\,E(x\ x)))$	$\{$ *β-reduint*$\}$
4		$=$	$E\ (\mathbf{Y}E)$	$\{$ *per igualtat entre 0 i 2*$\}$

Fixem-nos que per comprovar que funcionava com volíem hem fet un pas via igualtat, no via reducció. Amb aquest altre podem comprovar que funciona correctament només amb reducció.

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

Exercici 6 *Demostreu que* **T** *és realment un operador de punt fix.*

4.2.2 El Factorial

Mitjançant aquests operadors som capaços de definir funcions recursivament, vegem-ne el clàssic exemple del factorial. Una definició recursiva del factorial podria ser:

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

que es pot escriure com a:

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

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:

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

Fem els primers passos per a comprovar que realment funciona amb el factorial de 2, **fact 2** (per simplicitat farem definicions locals durant la derivació):

0	**fact 2**	=		
			$\overbrace{\qquad\qquad\qquad\qquad}^{\mathbf{F}}$	
1		=	$\mathbf{T}\,\overbrace{(\lambda f.\lambda n.\ (\ \mathbf{eszero}\ \ n\ \ \rightarrow \mathbf{1} \mid n * f\ (\mathbf{prec}\ \ n)))}\ \mathbf{2}$	{ *def. de* **fact**}
2		=	$(\lambda x.\,(\lambda y.\,y(x\ x\ y)))\ (\lambda x.\,(\lambda y.\,y(x\ x\ y)))\,\mathbf{F}\ \ \mathbf{2}$	{ *def. de* **T**}
2		=	$(\lambda y.\ y((\lambda x.\ (\lambda y.\ y(x\ x\ y)))\ (\lambda x.\ (\lambda y.\ y(x\ x\ y)))\ y))\,\mathbf{F}\ \ \mathbf{2}$	{ *β-reduint*}
			$\overbrace{\qquad\qquad\qquad\qquad\qquad\qquad}^{\mathbf{fact}}$	
3		=	$\mathbf{F}\,\overbrace{((\lambda x.\ (\lambda y.\ y(x\ x\ y)))\ (\lambda x.\ (\lambda y.\ y(x\ x\ y)))\ \mathbf{F})}\ \mathbf{2}$	{ *β-reduint*}
4		=	$(\lambda f.\lambda n.\ (\ \mathbf{eszero}\ \ n\ \ \rightarrow \mathbf{1} \mid n * f\ (\mathbf{prec}\ \ n)))\ \mathbf{fact\ 2}$	{ *def. de* **F**}
5		=	$\lambda n.\ (\ \mathbf{eszero}\ \ n\ \ \rightarrow \mathbf{1} \mid n * \mathbf{fact}\ (\mathbf{prec}\ \ n))\ \mathbf{2}$	{ *β-reduint*}
6		=	$(\ \mathbf{eszero}\ \ \mathbf{2} \rightarrow \mathbf{1} \mid \mathbf{2} * \mathbf{fact}\ (\mathbf{prec}\ \ \mathbf{2}))$	{ *β-reduint*}
7		=	$\mathbf{2} * \mathbf{fact}\ (\mathbf{prec}\ \ \mathbf{2})$	{ *def. condicio.* }
8		=	$\mathbf{2} * \mathbf{fact\ 1}$	{ *def.* **prec** }
9		=	$\mathbf{2} * \mathbf{T}\ (\lambda f.\lambda n.\ (\ \mathbf{eszero}\ \ n\ \ \rightarrow \mathbf{1} \mid n * f\ (\mathbf{prec}\ \ n)))\ \mathbf{1}$	{ *def.* **fact** }
10		=	$\ldots$	

Exercici 7 *Definiu la multiplicació recursivament. Comproveu que efectivament* 2×2 *és 4.*

Utilitzant el punt fixe podrem definir funcions per recursivitat. Si bé és farregós treballar amb λ-expressions, val a dir que els llenguatges de programació funcionals serien l'endolciment d'aquest model de còmput.

5 Assignació de Tipus

Fins ara el λ-càlcul que hem vist és el que es coneix com a *type-free theory*. Qualsevol expressió (considerada com a funció) es pot aplicar a qualsevol altra expressió (considerada com a argument). Com a exemple paradigmàtic, $(\lambda x.\, x)(\lambda x.\, x)$.

També hi ha però, versions del λ-càlcul amb tipus que vàren ser definides per Haskell B. Curry (1934) i per Alonzo Church (1940). Els tipus són objectes de natura sintàctica que es poden associar a λ-termes, per exemple si un tipus T s'assigna a un terme M, direm "*el terme M te tipus T*" i habitualment es denota com a $M \ : \ T$ (en HASKELL els tipus s'indiquen mitjançant `::`). Els tipus es construeixen a partir d'un conjunt de *tipus bàsics* o *atòmics*, d'un conjunt de variables de tipus $\mathcal{V}$ i del *constructor de tipus* funcional que és l'operador binari $\rightarrow$. Per exemple si volem representar el tipus d'una funció `f` que rep un enter i retorna un enter ho podem fer així: `f` : `Int` $\rightarrow$ `Int`. La noció de tipus aporta una component semàntica al càlcul que pot servir "metafòricament"per evitar barrejar *peres* i *pomes*...

L'estil que seguirem nosaltres és el de Curry que també es coneix com a sistema de tipus implícit i consistirà en un sistema d'assignació o associació de tipus per als λ-termes que coneixem. D'alguna manera aquest sistema correspon al que segueix HASKELL quan permet definir funcions i disposa d'un sistema d'inferència de tipus.

5.1 Tipus (a la 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:

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

on els tipus bàsic de $\mathcal{B}$ denoten certs conjunts (enters, booleans), on les variables de tipus $a \in \mathcal{V}$ denotes qualsevol tipus i el tipus $\tau_1 \ \rightarrow \ \tau_2$ denota el tipus de les funcions que van d'elements de tipus τ_1 a element de tipus τ_2. A més, l'operador $\rightarrow$ el considerarem associatiu a la dreta.

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 $\tau_1 = \tau_2$.

Les *derivacions* dels tipus es construiran a partir d'assumpcions com

$x : \tau$ amb les dues regles següents:

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

Direm que una assignació de tipus $M : \tau$ és derivable de Γ:

$$\Gamma \vdash M : \tau$$

si podem construïr una derivació de $M : \tau$ on totes les assumpcions que no estiguin cancel.lades (premises sense caixa) apareguin a Γ. En particular, quan no necessitem cap premisa ho notarem com a $\vdash M : \tau$.

Per exemple: sigui $\tau \in \mathcal{T}$ llavors $\vdash \lambda f, x.ff(x) : (\tau \to \tau) \to \tau \to \tau$ com mostra la següent derivació:

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

Aquest sistema te bones propietats (que no demostrarem):

1. El tipus d'un λ-terme es preserva encara que β-reduïm. Si $M \to_\beta M'$ llavors:
$$\Gamma \vdash M : \tau \Rightarrow \Gamma \vdash M' : \tau$$
compte, al revés no, és a dir si $M \to_\beta M'$ llavors:
$$\Gamma \vdash M' : \tau \not\Rightarrow \Gamma \vdash M : \tau$$

2. Qualsevol terme al qual puguem assignar un tipus (és a dir, que sigui *tipificable*) és *fortament normalitzant*: qualsevol seqüència de reduccions que fem a un terme tipificable és finita. Això no és pas cert per al λ-càlcul sense tipus (redordeu $(\lambda x.xx)(\lambda x.xx)$), de fet quin tipus li posarieu?...). Això implica que la forma normal de tot terme és computable i per tant la ïgualtat"entre termes tipificables és decidible. Com a conseqüència tot programa escrit amb termes tipificables acaba. Queda clar doncs, que aquest llenguatge (λ-càlcul tipat a la Curry) no és Turing-complet.

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. Fixeu-vos que un terme pot tenir més d'un tipus. D'altra banda, que un terme M tingui tipus no vol dir que si l'apliquen a un altre terme N, MN sigui tipificable...

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 $\vdash M : \tau$?

Fixeu-vos que de tots els elements que hem definit amb λ-càlcul sense tipus, només una petita part es tipificable... Concretament, tot terme que tingui "auto-aplicacions" com $\lambda x.xx$. Fixeu-vos per tant que els combinadors de punt fixe que hem vist no tenen tipus!!! Hem afegit tipus però hem perdut expressivitat com ara la recursivitat...

6 λ-càlcul Polimòrfic

El λ-càlcul tipat a la Curry te també varies extensions. Una d'elles és el λ-càlcul polimòrfic ($\lambda 2$). Intuïtivament el que aporta aquesta extensió és la possibilitat de fer assignacions de tipus *quantificades universalment*, és a dir, polimòrfiques, així per exemple, en $\lambda 2$ el tipus de $\lambda x.x$ seria $\forall \tau.\tau \to \tau$.

Malauradament, per a $\lambda 2$ es perd la decidibilitat de l'habitabilitat, mentres que la decidibilitat de la tipificabilitat i de la comprovació de tipus romanen com a problemes oberts...