CAP - Seqüències *Lazy*

Jordi Delgado, Gerard Escudero,

Tema 6

Avaluació mandrosa (*lazy*) vs. Avaluació Ansiosa (*eager*)

Clojure **NO** és un llenguatge amb avaluació mandrosa.

Quan s'invoca una funció en Clojure, els seus paràmetres s'avaluen abans de la crida. En aquest context, podríem dir que el contrari de *mandrós* (*lazy*) és *ansiós* (*eager*). Així doncs, Clojure és un llenguatge amb avaluació *ansiosa*.

Els llenguatges de programació amb avaluació mandrosa, p.ex. Haskell, només avaluen els arguments d'una funció si cal. Per exemple:

```
f = \x y -> x

f 2 3     👉 2
f 2 (1/0) 👉 2
```

En canvi, en Clojure:

```
(defn f [x y] x)

(f 2 3)       👉 2
(f 2 (/ 1 0)) 👉 Execution error (ArithmeticException)
```

El que sí fa Clojure és permetre utilitzar ***lazy sequences***.

Lazy Sequences

Una ***lazy sequence*** (***seqüència mandrosa*** (!?)) és aquella en que els elements de la seqüència no estan disponibles amb antelació i es produeixen com a resultat d'un càlcul. El càlcul es realitza segons calgui. L'avaluació de seqüències mandroses es coneix com a ***realització***.

Cal considerar un aspecte subtil: Parlem de *lazy* ***sequences*** perquè les seqüències són les úniques que són *lazy*. Els vectors no ho són. Els diccionaris no ho són. D'aquí que moltes de les funcions d'ordre superior que hem vist retornin seqüències.

Les funcions de la *Clojure sequence library* (com per exemple `map`, `filter`, `concat`, `take`, etc.) són ***lazy-friendly***: Això vol dir que si reben com a argument una seqüència *lazy* no imposen la seva realització, i que generen seqüències *lazy* (fins i tot si han rebut un vector o una col·lecció no-*lazy* com a argument).

També generen seqüències *lazy* aquelles funcions que retornen seqüències "infinites" (`iterate`, `repeat`, `range`, etc.)

Finalment, els nostres programes poden generar seqüències *lazy* fent servir constructors de seqüències *lazy*: `lazy-seq` o `lazy-cat`, per exemple.

Lazy Sequences

El fet que la *Clojure sequence library* tingui funcions *lazy-friendly* té conseqüències importants. Fins ara hem fet servir aquestes funcions ignorant la seva *laziness* implícita. Cal, però, que coneguem finalment quin és el funcionament real d'aquestes funcions.

Veiem alguns exemples (haver d'escriure -*print*- les seqüències *lazy* al REPL fa que es realitzin):

```
;; Multipliquem per 2 a tots els naturals. Obviament és lazy
;;
(def sl1 (map #(* % 2) (iterate inc 1)))

;; Aquesta seqüència també és lazy, encara que no sigui "infinita"
;;
(take 15 sl1) 👉 (2 4 6 8 10 12 14 16 18 20 22 24 26 28 30)

;; El resultat és lazy, encara que l'argument no ho sigui
;;
(def sl2 (map inc [1 2 3 4])

;; Aquest resultat ja NO és lazy, tot i que l'argument sí que ho és
;;
(vec sl2) 👉 [2 3 4 5]

;; Obtenir 10 nombres enters aleatoris entre 0 i 99
;;
(take 10 (repeatedly #(rand-int 100))) 👉 (80 52 30 29 10 52 39 69 22 53)
```

Lazy Sequences

Gràcies als efectes colaterals (*side effects*) és fàcil veure que les funcions que ja coneixem de la *Clojure sequence library* generen en realitat seqüències *lazy*.

```
user=> (def result (map println [:a :b :c]))
#'user/result
user=>
```

Com? No s'ha aplicat la funció `println` als elements del vector `[:a :b :c]`?

El que ha passat és que la seqüència resultant del `map` és *lazy*, per tant no es realitzarà mentre no faci falta. Si forcem la realització:

```
user=> result
👁  :a
👁  :b
👁  :c
👉 (nil nil nil)

user=> result
👉 (nil nil nil)
```

Ara s'ha executat `println` per a cada un dels elements de `[:a :b :c]`. `(nil nil nil)` és resultat de que el valor de retorn de `println` és `nil`.

Lazy Sequences: doall

`doall` serveix per realitzar una seqüència *lazy*.

Veiem-ne exemples:

En aquest exemple, `take` rep una seqüència *lazy* i retorna una seqüència *lazy*, per tant no s'executa res fins que no es realitza:

```
(let [x (atom 0)]
   (take 10 (repeatedly #(swap! x inc)))
   @x) 👉 0
```

`doall` serveix per forçar la realització de la seqüència:

```
(let [x (atom 0)]
   (doall (take 10 (repeatedly #(swap! x inc))))
   @x) 👉 10
```

Un altre exemple, `range` és *lazy*:

```
(time (nth (range 1 100000000 4) 16)) 👉 65
👁  "Elapsed time: 0.017652 msecs"
(time (nth (doall (range 1 100000000 4)) 16)) 👉 65
👁  "Elapsed time: 1686.357028 msecs"
```

Font: clojuredocs.org/clojure.core/doall

Lazy Sequences: doall

`doall` serveix per realitzar una seqüència *lazy*.

El podem fer servir per fer alguns experiments interessants:

```
(time (doall (map (fn [x]
                    (loop [i 1] (if (> i 1E+9) 1 (recur (inc i)))))
                  (range 10))))
👁  "Elapsed time: 8746.204096 msecs" ;; 8 segons!
👉 (1 1 1 1 1 1 1 1 1 1)
```

Com hem dit abans, `map` retorna una seqüència *lazy*, per tant fixem-nos que si no realitzem el resultat...

```
(time (def no-realitzada
           (map (fn [x]
                  (loop [i 1] (if (> i 1E+9) 1 (recur (inc i)))))
                (range 10))))
👁  "Elapsed time: 0.038851 msecs"
👉 #'user/no-realitzada

no-realitzada 👉 (1 1 1 1 1 1 1 1 1 1) ;; i ara sí que ha trigat 8 segons
```

... no s'inverteix temps en cap càlcul. Senzillament es construeix una seqüència *lazy* que està *pendent* de realització.

Lazy Sequences: doall

`doall` serveix per realitzar una seqüència *lazy*.

El podem fer servir per fer alguns experiments interessants:

En aquest experiment podem veure que les seqüències *lazy* en Clojure es guarden en memòria (són *cached*):

```
(let [s (time (map #(Thread/sleep %) (range 0 150 10)))]
  (time (doall s))
  (time (doall s)))
👉 (nil nil nil nil nil nil nil nil nil nil nil nil nil nil nil)

👁️ "Elapsed time: 0.029905 msecs"
👁️ "Elapsed time: 1903.846528 msecs"
👁️ "Elapsed time: 0.021917 msecs"
```

Si ens fixem, el segon cop que realitzem `s` no es triguen els 1.9 segons que triguem el primer cop. El resultat de realitzar `s` s'ha guardat.

Nota: Hem fet servir `Thread/sleep` només per provocar un alentiment de l'execució del codi. Aquesta funció, sense entrar en detalls, fa que el *thread* s'aturi un cert nombre de milisegons (aproximadament).

Lazy Sequences: lazy-seq

En Clojure es poden fer funcions que generin seqüències *lazy* amb `lazy-seq`.

Per exemple, podríem fer la nostra versió de `iterate` i utilitzar-la en diversos exemples:

```
(defn my-iterate [f x] ;; f ha de ser pura
  (lazy-seq
    (cons x (my-iterate f (f x)))))  ;; no hi ha cas base

(def powers-of-two (my-iterate (partial * 2) 1))
(nth powers-of-two 10) 👉 1024

;; Recordeu l'exemple de l'aproximació a l'arrel quadrada del tema 1??
;; Sabent (més) Clojure, ara l'escriurem així...

(defn arrel [x]
  (letfn [(arrels [x] (my-iterate #(float (/ (+ % (float (/ x %))) 2)) x))]
    (nth (arrels x) 100))) ;; suposem que 100 termes són suficients

(arrel 25)  👉  5.0
(arrel 36)  👉  6.0
(arrel 100) 👉 10.0
```

Fixem-nos que ja no és un problema fer definicions recursives *sense cas base* (!)

Lazy Sequences: lazy-seq

Durant el curs hem fet les nostres versions de `map`, `filter`, etc. per il·lustrar diverses tècniques. Naturalment no hem tingut en compte com de *lazy-friendly* són aquestes funcions. Fent servir `lazy-seq` sí que podem fer-ne una implementació pròpia i fidel a l'original:

```
(defn my-filter [pred coll]
  (let [step (fn [p c]
               (when-let [s (seq c)]
                 (if (p (first s))
                   (cons (first s) (my-filter p (rest s)))
                   (recur p (rest s)))))]
    (lazy-seq (step pred coll))))

(def s (my-filter #(when (even? %) (do (println %) %)) (range 100))) 👉 #'user/s
(take 5 s) 👉 ;; es confonen l'output de la crida a take i els println
(0
2
0 4
2 6
4 8
6 8)
;; però el segon cop ja hem realitzat aquest tros de la seqüència
(take 5 s) 👉 (0 2 4 6 8)
```

Lazy Sequences: lazy-seq

Els autors de *The Joy of Clojure* ens diuen*:

(...) the `lazy-seq` *recipe for applying laziness to your own functions:*

1. *Use the* `lazy-seq` *macro at the outermost level of your lazy sequence–producing expression(s).*
2. *If you happen to be consuming another sequence during your operations, then use* `rest` *instead of* `next`*.*
3. *Prefer higher-order functions when processing sequences.*
4. *Don't hold on to your head.*

Tots aquests punts s'entenen força bé... excepte el 4rt: Què vol dir *Don't hold on to your head*?

*Font: *The Joy of Clojure*, p. 126

Lazy Sequences: lazy-seq

A què es refereixen amb l'expressió *Don't hold on to your head*?

If you manage to hold on to the head of a sequence somewhere within a function, then that sequence will be prevented from being garbage collected. The simplest way to retain the head of a sequence is to bind it to a local.

Font: *The Joy of Clojure*, p. 128

La idea és que hem d'evitar de retenir el primer element d'una seqüència ja que això impedirà, mentre es processa la resta de la seqüència, que el que no es necessita sigui *garbage collected*:

```
;; Recordem que els rangs són lazy

(let [r (range 1e8)] (first r) (last r)) 👉 99999999

(let [r (range 1e8)] (last r) (first r)) 👉 Execution error (OutOfMemoryError)
                                           Java heap space
```

En el primer cas, Clojure sap que no cal retenir el `first` per calcular el `last`, així que tot el que cal generar per arribar al `last` pot ser *garbage collected*.

Això no passa en el segon cas. En voler retenir tot allò que cal generar per arribar al `last`, de cara a poder retornar el `first`, esgotem el *heap* de Java 😆

Lazy Sequences: dorun

`dorun` serveix per realitzar una seqüència *lazy*.

Però..., hem escrit això també per a `doall`. Quina és la diferència?

- `doall` - Guarda en memòria la seqüència mentre força la realització dels elements. Retorna la seqüència.
- `dorun` - No guarda la seqüència en memòria mentre força la realització dels diversos elements. Retorna `nil`.

Podríem il·lustrar la diferència amb una implementació (simplificada):

```
(defn dorun [coll]
  (when (seq coll) (recur (next coll))))
```

```
(defn doall [coll]
  (dorun coll) coll)
```

```
user=> (doall (map #(do (println %) %) [1 2 3]))
👁  1
👁  2
👁  3
👉 (1 2 3)  ;; valor de retorn
user=> (dorun (map #(do (println %) %) [1 2 3]))
👁  1
👁  2
👁  3
👉 nil      ;; valor de retorn
```

Lazy Sequences: delay & force

Malgrat la *laziness* de les seqüències a Clojure, pot passar que vulguem tenir més control sobre l'avaluació d'expressions. Les macros `delay` i `force` serveixen precisament per a això.

- `(delay & expressions)`: Pren un grup d'expressions i produeix un objecte que invocarà les expressions només la primera vegada que forcem la seva avaluació (amb `force` o `deref/@`). Aquest resultat es guarda en una *cache* i serà retornat, sense re-avaluar les expressions, les següents vegades que es demani l'avaluació del `delay`.

- `(force x)` Si `x` és resultat d'un `delay` anterior, `force` retorna el resultat d'avaluar el `delay`, possiblement retornant l'objecte en la *cache*

```
user=> (def my-delay (delay (println "escric al terminal") 100))
👉 #'user/my-delay
user=> (force my-delay)
👁️ escric al terminal
👉 100
user=> (force my-delay)  ;; no torna a executar, s'escriu la cache
👉 100
```

Lazy Sequences: delay & force

Per exemple, suposem que volem una seqüència *lazy* dels nombres triangulars:

```
(defn triangle [n] (/ (* n (+ n 1)) 2)) ;; nombres triangulars
(def tri-nums (map triangle (iterate inc 1)))

(take 10 tri-nums) 👉 (1 3 6 10 15 21 28 36 45 55)
(take 10 (filter even? tri-nums)) 👉 (6 10 28 36 66 78 120 136 190 210)
```

Podem fer-ho de manera alternativa amb `delay`/`force`:

```
(defn inf-triangles [n]
  {:head (triangle n) :tail (delay (inf-triangles (inc n)))})

(defn head [l] (:head l))
(defn tail [l] (force (:tail l)))

(def tri-nums (inf-triangles 1))

(head tri-nums) 👉 1
(head (tail tri-nums)) 👉 3
(head (tail (tail tri-nums))) 👉 6
```

(continua ➡️)

Font: *The Joy of Clojure*, p. 129

Lazy Sequences: delay & force

Podem fer servir `head` i `tail` com a primitives per fer funcions que permetin tenir accés als elements de la seqüència de la mateixa manera que teníem amb les seqüències *lazy*.

Per exemple:

```
(defn taker [n l]
  (loop [t n, src l, ret []]
    (if (zero? t)
      ret
      (recur (dec t) (tail src) (conj ret (head src))))))

(taker 10 tri-nums) 👉 [1 3 6 10 15 21 28 36 45 55]

(defn nthr [l n]
  (if (zero? n)
    (head l)
    (recur (tail l) (dec n))))

(nthr tri-nums 99)  👉 5050
(nthr tri-nums 110) 👉 6216
```

Exercici: Proposeu una manera alternativa de fer quasi bé el mateix que la parella `delay`/`force`. Diem *quasi bé* ja que no demanem que es memoritzi el resultat. Modifiqueu la vostra proposta per a que ara sí es memoritzi el resultat.

Font: *The Joy of Clojure*, p. 129

Lazy Sequences: Seqüències "infinites"

La possibilitat d'escriure programes com a accessos *finits* a seqüències *"infinites"* afegeix força expressivitat al llenguatge.

Per exemple, si volem tenir els primers k múltiples d'un nombre n (problema trivial per altra part) podem fer:

```
(def multiples
  (for [n (iterate inc 2)] (map #(* n %) (iterate inc 1)))) 👉 #'user/multiples

;; i ara en tinc prou fent (suposem que k és 10 i n és 154):

(take 10 (nth multiples 152)) 👉 (154 308 462 616 770 924 1078 1232 1386 1540)
```

Podem implementar un *merge lazy* per poder treballar amb seqüències *lazy*:

```
(defn my-merge [[cap1 & cua1 :as lst1] [cap2 & cua2 :as lst2]]
  (lazy-seq
   (cond
     (< cap1 cap2) (cons cap1 (my-merge cua1 lst2))
     (= cap1 cap2) (cons cap1 (my-merge cua1 cua2))   ;; elimina duplicats
     (> cap1 cap2) (cons cap2 (my-merge lst1 cua2)))))

(def p (my-merge (range) (range -50 100 10))) 👉 #'user/p
(take 10 p) 👉 (-50 -40 -30 -20 -10 0 1 2 3 4)
```

Lazy Sequences: mapv / filterv

`mapv` i `filterv` són les versions no-*lazy* de `map` i `filter`. Són no-*lazy* ja que retornen un vector. Ja hem dit abans que els vectors en Clojure no són *lazy*.

Tornant a l'exemple de `map` amb `println`:

```
user=> (def result (map println [:a :b :c]))
👉 #'user/result
user=> (def result (mapv println [:a :b :c]))
👁️ :a
👁️ :b
👁️ :c
👉 #'user/result
```

A banda d'aquest fet, el seu comportament és el mateix que les seves contrapartides *lazy*.

```
;; filter retorna seqüència
;;
(filter #(= (count %) 1) ["a" "aa" "b" "n" "f" "lisp" "clojure" "q" ""])
👉 ("a" "b" "n" "f" "q")

;; filterv retorna vector
;;
(filterv #(= (count %) 1) ["a" "aa" "b" "n" "f" "lisp" "clojure" "q" ""])
👉 ["a" "b" "n" "f" "q"]
```

Lazy Sequences: Sequence Chunking

En un intent de fer l'ús de seqüències *lazy* més eficient, Clojure fa el que s'anomena ***sequence chunking***. Això vol dir que quan es demana la realització d'elements de seqüències *lazy*, això es fa en blocs de 32 elements:

```
;; |--- Només demano el primer element de la seqüència lazy!
;; v
(first (map #(do (print % " ") (identity %)) (range 100)))
👁️ 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 2
👉 0
```

Hem demanat el primer element de la seqüència, per tant només calia que es realitzés aquest. Fixem-nos, però, que s'han realitzat els 32 primers, és a dir, s'ha calculat el resultat d'aplicar la funció `#(do (print % " ") (identity %))` als 32 primers elements de `(range 100)`).

És una manera d'*amortitzar* el cost -per element- de generar elements. Per exemple, fixem-nos que si demanem 33 elements, acabem realitzant-ne 64.

Lazy Sequences: Exercicis Recapitulatoris

- Implementa `my-map`, ara tenint en compte la *laziness* del `map` original.
- Fes una funció per generar la seqüència *ordenada* dels **nombres de Hamming**: [1,2,3,4,5,6,8,9,...]. Els nombres de Hamming són aquells que tenen només 2, 3, i/o 5 com a divisors primers.

```
;; si la seqüència s'anomena hammings...
(def hammings ...)
(take 15 hammings) 👉 (1 2N 3N 4N 5N 6N 8N 9N 10N 12N 15N 16N 18N 20N 24N)
```

- Donat un nombre qualsevol de seqüències no buides, cadascuna ordenada de més petit a més gran, trobeu el nombre més petit que apareix a totes les seqüències (suposeu que aquest existeix). Les seqüències poden ser infinites, així que aneu amb compte i feu cerques *lazy*.

```
;; si la funció s'anomena cerca-lazy
(cerca-lazy (map #(* % % %) (range)) (iterate inc 20)) 👉 27
```

Lazy Sequences: Exercicis Recapitulatoris

- En el que segueix, `m`, `n`, `s`, `t` denoten nombres enters no negatius, `f` denota una funció que accepta dos arguments i es defineix per a tots els nombres enters no negatius dels dos arguments. En matemàtiques, la funció `f` es pot interpretar com una matriu infinita amb infinites files i columnes que, quan s'escriu, sembla una matriu ordinària, però les seves files i columnes no es poden escriure completament, de manera que s'acaben amb el·lipses. A Clojure, aquesta matriu infinita es pot representar com una seqüència *lazy* infinita de seqüències *lazy* infinites, on les seqüències interiors representen files. Escriu una funció que accepti 1, 3 i 5 arguments:
  - amb l'argument `f`, retorna la matriu infinita `A` que té l'entrada a la fila `i` i la columna `j` és igual a `f(i,j)` per a `i, j = 0,1,2,...`
  - amb els arguments `f`, `m`, `n`, retorna la matriu infinita `B` que és igual a la resta de la matriu `A` després de l'eliminació de les primeres `m` files i les primeres `n` columnes
  - amb els arguments `f`, `m`, `n`, `s`, `t`, retorna la matriu finita `s`$\times$`t` que consta de les primeres `t` entrades de cadascuna de les primeres `s` files de la matriu `B` o, de manera equivalent, que consta de la primeres `s` entrades de cadascuna de les primeres `t` columnes de la matriu `B`.

```
;; si la funció s'anomena imat
(imat * 3 5 5 7) 👉 [[15 18 21 24 27 30 33]
                     [20 24 28 32 36 40 44]
                     [25 30 35 40 45 50 55]
                     [30 36 42 48 54 60 66]
                     [35 42 49 56 63 70 77]]
```

Lazy Sequences

Per saber-ne més:

- *The Joy of Clojure*, seccions 6.3, p. 123, i 6.4, p. 132
- *Language: Laziness*

Per a una visió crítica de la *laziness* i el seu ús a Clojure:

- *Clojure's Deadly Sin*, entrada del blog d'Oleksandr Yakushev el 27 de Juliol de 2023.