PARADIGME DE PROGRAMARE

Curs 9

Fluxuri.

Fluxuri – Cuprins

- Motivație
- Fluxuri
- Fluxuri definite explicit
- Fluxuri definite implicit
- Ciurul lui Eratostene
- Utilizări

Modelarea lumii înconjurătoare

Scopul sistemelor software = modelarea lumii prin reducerea complexității:

- Tehnice: timp, spațiu
- Intelectuale: abstractizarea modulelor în spatele unor interfețe simplu de utilizat

Paradigme de modelare

- Imperativă: Obiecte (module) care își schimbă starea în timp
  - Obiecte ca module: starea este înglobată în interiorul obiectelor (ex: obiectul „promisiune")
  - Efecte laterale: probleme de sincronizare și partajare
  - Schimbare în timp: timpul din program este strâns legat de timpul real, stările vechi se pierd
- Funcțională: Un tot imuabil descris prin colecția stadiilor sale de evoluție (ex: funcțiile sin, cos)
  - Funcții ca module: funcții independente („cutii negre"), care pot fi analizate/testate izolat
  - Puritate: funcții matematice, fără efecte laterale
  - Evoluție în timp: funcțiile construiesc stări noi pe baza celor vechi, stările anterioare rămân accesibile

Exemplu la calculator

Ex: Să se determine dacă un număr n este prim.

Definiție

Un număr n este prim dacă și numai dacă nu are divizori în intervalul [2 .. √n].

O soluție modulară

Eleganță versus Eficiență

```
1.  (define (interval a b)
2.    (if (> a b)
3.        '()
4.        (cons a (interval (add1 a) b))))
5.  (define (prime? n)
6.    (null?
7.     (filter (λ (d) (zero? (modulo n d)))
8.             (interval 2 (sqrt n)))))
```

- Ineficient spațial: Reține tot intervalul
- Ineficient temporal: Procesează tot, chiar când ar putea găsi instant un divizor
- Elegant: separă clar modulele (conform diagramei)

```
1.  (define (prime? n)
2.    (let iter ((div 2))
3.      (cond
4.        ((> (* div div) n) #t)
5.        ((zero? (modulo n div)) #f)
6.        (else (iter (add1 div))))))
```

- Eficient spațial: Reține doar 2 variabile (n, div)
- Eficient temporal: Se oprește la primul divizor
- Neelegant: amestecă generarea / filtrarea / testarea

Fluxuri – Cuprins

- Motivație
- Fluxuri
- Fluxuri definite explicit
- Fluxuri definite implicit
- Ciurul lui Eratostene
- Utilizări

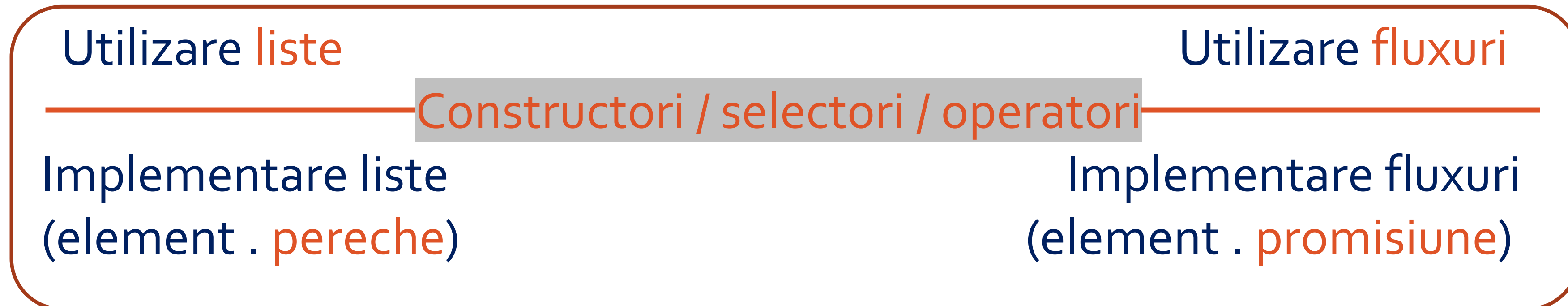
Eleganță plus Eficiență

```
1.  (define (interval-stream a b)
2.    (if (> a b)
3.        empty-stream
4.        (stream-cons a (interval-stream (add1 a) b))))
5.  (define (prime? n)
6.    (stream-empty?
7.     (stream-filter (λ (d) (zero? (modulo n d)))
8.                    (interval-stream 2 (sqrt n)))))
```

interfață foarte asemănătoare celei pentru liste

- Elegant și eficient temporal și spațial, deși, conceptual, este același program cu cel pe liste
- Care este diferența esențială între varianta cu liste și varianta cu fluxuri?

Sub bariera de abstractizare



Complexitate spațială

- Intervalul [1 .. n] reținut ca listă: Θ(n) (n elemente ținute în memorie)
- Intervalul [1 .. n] reținut ca flux: Θ(1) (un element și o promisiune de a evalua restul fluxului)

1

```
(delay (interval-stream 2 b))
```

(rețin textul expresiei +
contextul – ce valori referă interval-stream și b)

Complexitate temporală

- (prime? n) cu liste: Θ(√n) (generez √n numere, parcurg √n numere ca să le filtrez, aplic null?)
- (prime? n) cu fluxuri: Θ(distanța până la primul divizor) (Θ(√n) pentru n prim)

Exemplu

```
(prime? 115)
(stream-empty? (stream-filter div? '(2 . promise[3-10.72])))
      ;; 115 ⋮ 2?   Nu, atunci filtrăm restul intervalului
(stream-empty? (stream-filter div? '(3 . promise[4-10.72])))
      ;; 115 ⋮ 3?   Nu, atunci filtrăm restul intervalului
(stream-empty? (stream-filter div? '(4 . promise[5-10.72])))
      ;; 115 ⋮ 4?   Nu, atunci filtrăm restul intervalului
(stream-empty? (stream-filter div? '(5 . promise[6-10.72])))
      ;; 115 ⋮ 5?   Da, atunci avem un rezultat pentru stream-filter
      ;; Acesta i-a cerut intervalului să se deșire până a găsit un divizor
(stream-empty? '(5 . promise[stream-filter div? '(6 . promise[7-10.72])]))
      ;; #f (stream-empty nu are nevoie să evalueze restul fluxului)
```

Liste versus fluxuri – Comparație

Criteriu	Listă	Flux (~listă cu evaluare întârziată)
Abstractizare (eleganță)	Interfață bine definită	Interfață bine definită
Construcție și prelucrare	Separate conceptual și computațional	Separate doar conceptual (intern, construcția este dirijată de nevoia de prelucrare)
Eficiență	Ineficientă: - stochează / procesează toate elementele	Eficient: - stochează / procesează doar atâtea elemente câte sunt necesare

Iluzie (întregul flux) versus realitate (evaluare parțială):

- Extinderea fluxului se realizează:
  - Automat: atât cât este necesar continuării execuției
  - Incremental: element cu element

Liste versus fluxuri – Interfață

Constructor pe liste	Constructor corespunzător pe fluxuri
'()	**empty-stream**
cons	**stream-cons**

Operator pe liste	Operator corespunzător pe fluxuri
null?	**stream-empty?**
car	**stream-first**
cdr	**stream-rest**
map	**stream-map**
filter	**stream-filter**

Fluxuri – Cuprins

- Motivație
- Fluxuri
- Fluxuri definite explicit
- Fluxuri definite implicit
- Ciurul lui Eratostene
- Utilizări

Definiții explicite (generând fiecare element)

Flux = pereche (element . motor-capabil-să-genereze-restul-fluxului)

Motor: implementat explicit ca o funcție recursivă,
cu parametri pe baza cărora să se poată genera elementul următor din flux

Exemple

```
(define naturals
  (let loop ((n 0))                ;; numărul curent în flux
    (stream-cons n (loop (add1 n)))))

(define factorials                 ;; n-ul următor
  (let loop ((n 1) (fact 1))       ;; și factorialul curent
    (stream-cons fact (loop (add1 n) (* n fact)))))
```

Extinderea fluxului la cerere

```
(define naturals
  (let loop ((n 0))
    (stream-cons n (loop (add1 n)))))
```

- Stare inițială: fluxul cunoaște doar primul element: naturals = ‘(0 …)
- Extindere la cerere: Calculul noilor elemente se face doar la `stream-rest`)
  - Ex: `(stream-take naturals 3)` solicită 2 elemente noi
  - ‘(0 …) (loop 1) ‘(0 1 …) (loop 2) ‘(0 1 2 …)
- Abstractizare: Fără știința utilizatorului, implementarea se bazează pe promisiuni:
  - `stream-cons` face un `delay`: `(stream-cons a b) ~ (cons a (delay b))`
  - `stream-rest` face un `force`: `(stream-rest s) ~ (force (cdr s))`

Fluxuri – Cuprins

- Motivație
- Fluxuri
- Fluxuri definite explicit
- Fluxuri definite implicit
- Ciurul lui Eratostene
- Utilizări

Definiții implicite (pe baza altor fluxuri)

Definiție implicită (fără generator (motor) explicit)

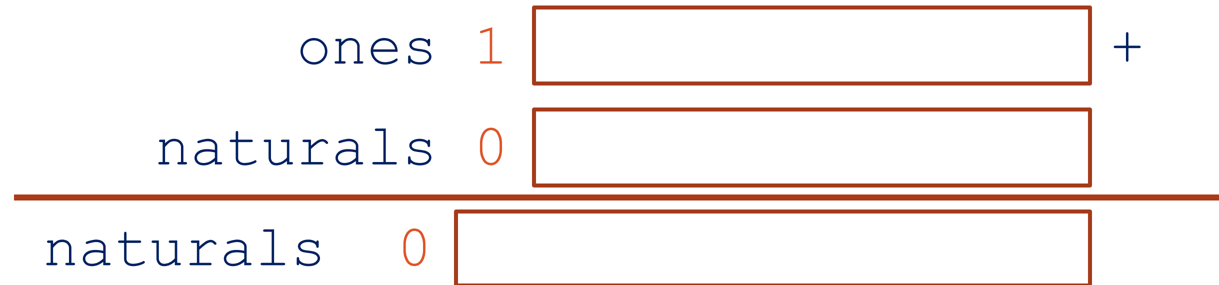
- Profită de evaluarea leneșă pentru a defini fluxul pe baza altor fluxuri (sau a lui însuși)
- Mecanisme uzuale
  - Transformarea/filtrare: map sau filter pe un flux existent
  - Combinare: operații între fluxuri existente (adunare, înmulțire, etc.)
  - Autoreferențiere: fluxul este definit pe baza lui însuși (și, opțional, pe baza altor fluxuri)
    - furnizăm explicit cel puțin un element; fără un punct de pornire, fluxul nu poate participa la o primă operație

Exemple

```
(define ones (stream-cons 1 ones))

(define naturals- (stream-cons 0 (stream-zip-with + ones naturals-)))
```

Operații între fluxuri – Funcționare

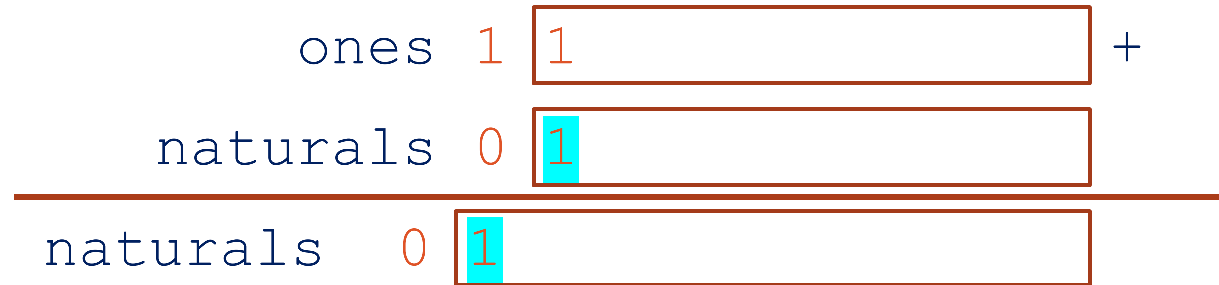


- `naturals[1]` are nevoie de `ones[0]` și `naturals[0]`, care sunt deja disponibile

- Observație: Pentru a putea începe adunările, am furnizat explicit `naturals[0]`:

```
(define naturals- (stream-cons 0 (stream-zip-with + ones naturals-)))
```

Operații între fluxuri – Funcționare



- `naturals[2]` are nevoie de `ones[1]` și `naturals[1]`, care sunt deja disponibile

Operații între fluxuri – Funcționare

```
ones      1 1 1 1 1 1 1 1 1 +
naturals  0 1 2 3 4 5 6 7 8
----------------------------
naturals  0 1 2 3 4 5 6 7 8 ......
```

- La fiecare pas, `ones` și `naturals` sunt evaluate exact cât este necesar pentru a calcula următorul element din `naturals`

Operații între fluxuri – Fibonacci

```
         fibo  0 | 1  1  2  3   5   8  |  +
(rest  fibo)  1 | 1  2  3  5   8  13  |
---------------------------------------------
fibo     ?     | 1  2  3  5  8  13  21  |......
```

- Pentru a începe adunările, ce trebuie furnizat explicit?

Operații între fluxuri – Fibonacci

```
          fibo  0 1 1 2 3  5  8   +
 (rest  fibo)  1 1 2 3 5  8 13
-----------------------------------
 fibo      0  1 1 2 3 5 8 13 21 .......
```

- Pentru a începe adunările, ce trebuie furnizat explicit?
- Observații:
  - Rezultat final: lipsesc termenii `0` și `1`
  - Prima operație: necesită `fibo[0]` și `fibo[1]` `(rest-fibo[0])`

  → furnizăm explicit primii 2 termeni

Mai multe definiții implicite

Exemple la calculator:

- Fluxul numerelor pare
- Fluxul puterilor lui 2
- Fluxul 1/n! – cu care se poate aproxima numărul e
- Fluxul sumelor parțiale ale altui flux (atenție la definiția eficientă versus cea ineficientă!)

Reutilizarea fluxului

Sub bariera de abstractizare: promisiuni versus închideri funcționale

- Eficiență: diferență uriașă în cazul reutilizării unui flux parțial evaluat
  - Promisiuni: nu reevaluează porțiunile deja calculate, ci iau rezultatele din cache
  - Închideri funcționale: reevaluează tot

Exemplu la calculator: fibonacci cu închideri versus fibonacci cu promisiuni

Fluxuri – Cuprins

- Motivație
- Fluxuri
- Fluxuri definite explicit
- Fluxuri definite implicit
- Ciurul lui Eratostene
- Utilizări

Ciurul lui Eratostene

2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 ...

- Stare inițială: fluxul numerelor naturale, începând cu 2
- Repetă:
  - Primul element p din flux este prim
  - Elimină multiplii lui p din flux

Ciurul lui Eratostene

2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 ...

- Stare inițială: fluxul numerelor naturale, începând cu 2
- Repetă:
  - Primul element p din flux este prim
  - Elimină multiplii lui p din flux

Ciurul lui Eratostene

2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 ...

- Stare inițială: fluxul numerelor naturale, începând cu 2
- Repetă:
  - Primul element p din flux este prim
  - Elimină multiplii lui p din flux

Ciurul lui Eratostene – Implementare

```
1.  (define (sieve s)
2.    (let ((p (stream-first s)))
3.      (stream-cons p (sieve (stream-filter
4.                             (λ (n) (not (zero? (modulo n p))))
5.                             (stream-rest s))))))
6.
7.  (define primes
8.    (sieve (stream-rest (stream-rest naturals))))
```

Fluxuri – Cuprins

- Motivație
- Fluxuri
- Fluxuri definite explicit
- Fluxuri definite implicit
- Ciurul lui Eratostene
- Utilizări

Utilizări

- Scurtcircuit: evitarea calculelor inutile
  - Ex: caută primul multiplu de 17 într-un flux (se oprește la primul)
- Modelare matematică: date nelimitate
  - Ex: fluxul numerelor prime, arbori de decizie (minimax), jocul „life"
- Pipelines: decuplarea modulelor
  - Ex: generare -> filtrare -> procesare pentru primele n elemente filtrate (generatorul nu trebuie să știe a priori de câte elemente va fi nevoie)
- Procesare în timp real: pe măsură ce apar noi date
  - Ex: monitorizează click-urile dintr-o zonă de ecran, redă cadre video pe măsură ce le primești
- Procesare de date „gigant": eficient spațial (întregul ca flux al părților sale)
  - Ex: caută un termen într-un fișier foarte mare, procesează un film ca flux de cadre

Rezumat

Flux

Implementare

Avantaj

Constructori

Operatori

Definire explicită

Definire implicită

Rezumat

Flux: secvență parțial construită (~listă cu evaluare întârziată)

Implementare

Avantaj

Constructori

Operatori

Definire explicită

Definire implicită

Rezumat

Flux: secvență parțial construită (~listă cu evaluare întârziată)

Implementare: pereche (element promisiune)

Avantaj

Constructori

Operatori

Definire explicită

Definire implicită

Rezumat

Flux: secvență parțial construită (~listă cu evaluare întârziată)

Implementare: pereche (element promisiune)

Avantaj: eleganță (modularitate), eficiență (temporală și spațială)

Constructori

Operatori

Definire explicită

Definire implicită

Rezumat

Flux: secvență parțial construită (~listă cu evaluare întârziată)

Implementare: pereche (element promisiune)

Avantaj: eleganță (modularitate), eficiență (temporală și spațială)

Constructori: empty-stream, stream-cons

Operatori

Definire explicită

Definire implicită

Rezumat

Flux: secvență parțial construită (~listă cu evaluare întârziată)

Implementare: pereche (element promisiune)

Avantaj: eleganță (modularitate), eficiență (temporală și spațială)

Constructori: empty-stream, stream-cons

Operatori: stream-empty?, stream-first, stream-rest, stream-map, stream-filter

Definire explicită

Definire implicită

Rezumat

Flux: secvență parțial construită (~listă cu evaluare întârziată)

Implementare: pereche (element promisiune)

Avantaj: eleganță (modularitate), eficiență (temporală și spațială)

Constructori: empty-stream, stream-cons

Operatori: stream-empty?, stream-first, stream-rest, stream-map, stream-filter

Definire explicită: generator recursiv care produce fluxul element cu element

Definire implicită

Rezumat

Flux: secvență parțial construită (~listă cu evaluare întârziată)

Implementare: pereche (element promisiune)

Avantaj: eleganță (modularitate), eficiență (temporală și spațială)

Constructori: empty-stream, stream-cons

Operatori: stream-empty?, stream-first, stream-rest, stream-map, stream-filter

Definire explicită: generator recursiv care produce fluxul element cu element

Definire implicită: transformarea/combinarea altor fluxuri (sau a fluxului însuși)