

KATEDRA INFORMATIKY, PŘÍRODOVĚDECKÁ FAKULTA  
UNIVERZITA PALACKÉHO, OLOMOUČ

# PARADIGMATA PROGRAMOVÁNÍ 2

## PROUDY A VYROVNÁVACÍ PAMĚŤ

Slajdy vytvořili Vilém Vychodil a Jan Konečný

## Implicitní definice nekonečného proudu

- definice proudu, která v sobě používá proud, který sama definuje
- *následující prvky proudu* jsou přímo zavedeny pomocí předchozích *prvků* proudu bez vytváření pomocné rekurzivní procedury (bez limitní podmínky)

nekonečný proud

```
(define pow2
  (let next ((last 1))
    (cons-stream last
                  (next (* 2 last )))))
```

implicitní definice předchozího

```
(define pow2
  (cons-stream 1
              (stream-map (lambda (x) (* 2 x))
                          pow2)))
```

## Ukázky implicitních definic proudů

proud faktoriálů

```
(define fak-stream
  (cons-stream 1
    (stream-map *
      fak-stream
      (stream-cdr naturals))))
```

proud Fibonacciho čísel

```
(define fib-stream
  (cons-stream 1
    (cons-stream 1
      (stream-map +
        fib-stream
        (stream-cdr fib-stream))))))
```

## Konstruktor nekonečných proudů

- nekonečné proudy lze vytvářet procedurou `build-stream`
- analogická procedura `build-list`, ale *nemá limitní podmínku* (není potřeba předávat „délku vytvářeného streamu“)

*;; konstruktor `build-stream`*

```
(define build-stream
  (lambda (f)
    (let proc ((i 0))
      (cons-stream (f i)
                    (proc (+ i 1))))))
```

### Příklady

```
(define ones (build-stream (lambda (i) 1)))
(define naturals (build-stream (lambda (i) (+ i 1))))
```

## Příklady manipulace s nekonečnými proudy

*;; vytváření nekonečných proudů z nekonečných proudů*  
*;; aplikací po sobě jdoucích funkcí na každý prvek*

```
(define expand-stream
  (lambda (stream . modifiers)
    (let next ((pending modifiers))
      (if (null? pending)
          (apply expand-stream
                 (stream-cdr stream) modifiers)
          (cons-stream ((car pending) (stream-car stream))
                       (next (cdr pending)))))))
```

*;; příklady použití*

```
(expand-stream ones - +)       $\implies$  proud: -1 1 -1 1 ...
(expand-stream ones + -)       $\implies$  proud: 1 -1 1 -1 ...
(expand-stream naturals + -)    $\implies$  proud: 1 -1 2 -2 3 ...
```

## Příklady manipulace s nekonečnými proudy

vytvoření proudu celých čísel

```
(define integers
  (build-stream (lambda (i)
                 (if (= i 0)
                     0
                     ((if (even? i) + -)
                      (quotient (+ i 1) 2))))))
```

nebo použitím streamu přirozených čísel a `expand-stream`

```
(define integers
  (cons-stream 0 (expand-stream naturals - +)))
```

v obou případech dostáváme

`integers`  $\implies$  proud: 0 -1 1 -2 2 -3 3 -4 4 -5 ...

## Příklady (nesprávné) manipulace s nekonečnými proudy

Následující má smysl pouze pokud je `s1` konečný:

```
(define stream-append2
  (lambda (s1 s2)
    (stream-foldr (lambda (x y)
                   (cons-stream x (force y)))
                  s2 s1)))
```

druhý proud se neuplatní, protože první je nekonečný

```
(stream-append ones (stream-map - ones))
```

následující bude cyklovat

```
(stream-length ones)
```

- počítat délku (nekončeného) streamu je „nesmysl“
- neexistuje algoritmus (a nikdy existovat nebude), který by pro daný stream rozhodl, zda-li je konečný či nikoliv

## Proud racionálních čísel

- můžeme vytvořit i proud všech racionálních čísel
- využijeme faktu, že všechna kladná racionální čísla jsou zapsána v následující tabulce (každé dokonce nekonečně mnoho krát) a toho, že tabulku můžeme projít po položkách v diagonálním směru

$\langle 1, 1 \rangle$	$\langle 1, 2 \rangle$	$\langle 1, 3 \rangle$	$\langle 1, 4 \rangle$	$\dots$
$\langle 2, 1 \rangle$	$\langle 2, 2 \rangle$	$\langle 2, 3 \rangle$	$\langle 2, 4 \rangle$	
$\langle 3, 1 \rangle$	$\langle 3, 2 \rangle$	$\langle 3, 3 \rangle$	$\langle 3, 4 \rangle$	$\dots$
$\langle 4, 1 \rangle$	$\langle 4, 2 \rangle$	$\langle 4, 3 \rangle$	$\langle 4, 4 \rangle$	
$\vdots$		$\vdots$		$\ddots$

**POZNÁMKA:** „proud všech reálných čísel“ nelze vytvořit (!)



## Proud racionálních čísel

;; proud párů (1 . 1) (2 . 1) (1 . 2) (3 . 1) ...

```
(define pairs
  (let next ((sum 2)
            (a 1)
            (b 1))
    (cons-stream (cons a b)
                 (if (= a 1)
                     (next (+ sum 1) sum 1)
                     (next sum (- a 1) (+ b 1))))))
```

;; proud zlomků 1/1 2/1 1/2 3/1 2/2 1/3 ...

```
(define fractions
  (stream-map (lambda (x) (/ (car x) (cdr x))) pairs))
```

kladná racionální čísla: zbývá odstranit opakující se čísla z `fractions`  
(například 1/1 je totéž jako 2/2, a podobně)

## Proud racionálních čísel

*;; proud kladných racionálních čísel*

*;; stejná čísla jsou odstraněna filtrací*

```
(define positive-rationals
  (let next ((f-stream fractions))
    (cons-stream
      (stream-car f-stream)
      (next (stream-filter
              (lambda (x)
                (not (= x (stream-car f-stream))))
              f-stream)))))
```

*;; a konečně: proud všech racionálních čísel*

*;; 0 -1 1 -2 2 -1/2 1/2 -3 3 -1/3 1/3 -4 4 -3/2 3/2 ...*

```
(define rationals
  (cons-stream 0 (expand-stream positive-rationals - +)))
```

## Kdy použít proudy?

### Kdy použít proudy místo seznamů?

- 1 když potřebujeme *řídít průběh výpočtu pomocí dat*
- 2 když *není dopředu známa velikost dat*, která chceme zpracovávat, nebo není možné odhadnout, kolik dat budeme muset zpracovat, než najdeme řešení (nějakého) problému
  - **Příklad:** pokud se budeme pokoušet najít ve velkém souboru sekvenci nějakých znaků odpovídající danému vzoru, pak nemá smysl natahovat celý vstupní soubor do paměti, což může být dlouhá operace (nebo i neproveditelná operace), protože hledaný řetězec může být třeba někde na začátku souboru.

### Typické použití proudů:

- řízení vstupně/výstupních operací
- práce se soubory
- používá mnoho PJ (například C++)
- my ukážeme implementaci V/V operací ve Scheme

## Vstupně/výstupní operace ve Scheme

- Scheme používá při manipulaci se soubory tzv. „porty“
- **port** lze chápat jako identifikátor otevřeného souboru
- pro detaily viz specifikaci R6RS

Příklad ukládání dat do souboru

```
(define p (open-output-file "soubor.txt"))  
(display "Ahoj svete!" p)  
(newline p)  
(display (map - '(1 2 3 4)) p)  
(newline p)  
(close-output-port p)
```

Příklad načítání dat ze souboru

```
(define p (open-input-file "soubor.txt"))  
(display (read p))  
(display (read p))  
(display (read p))  
(close-input-port p)
```

## Jednoduché vstupní proudy

*;; vytvoř proud čtením výrazů ze vstupního portu*

```
(define input-port->stream
  (lambda (reader port)
    (let iter ()
      (let ((elem (reader port)))
        (if (eof-object? elem)
            (begin
              (close-input-port port)
              '())
            (cons-stream elem (iter)))))))
```

*;; vytvoří proud otevřením souboru*

```
(define file->stream
  (lambda (reader file-name)
    (input-port->stream
     reader
     (open-input-file file-name))))
```

## Zvýšení výpočetní efektivity pomocí proudů

- predikát `equal-fringe?` je pro dva seznamy pravdivý p. k. oba seznamy mají stejné atomické prvky pokud je projdeme zleva-doprava

*;; přímočaré řešení, které je neefektivní*

```
(define equal-fringe?  
  (lambda (s1 s2)  
    (define flatten ; pomocná procedura: linearizace seznamu  
      (lambda (l)  
        (cond ((null? l) '())  
              ((list? (car l))  
               (append (flatten (car l))  
                        (flatten (cdr l))))  
              (else (cons (car l) (flatten (cdr l))))))  
      (equal? (flatten s1) (flatten s2))))
```

*;; příklad použití:*

```
(equal-fringe? '(a (b (c)) () d) '(a b c (d)))  $\implies$  #t  
(equal-fringe? '(a (b (c)) () d) '(a b c (e)))  $\implies$  #f
```

## Zvýšení výpočetní efektivity pomocí proudů

### V čem spočívá neefektivita předchozího řešení?

- 1 během výpočtu se konstruuje lineární seznamy, které se potom použijí pouze jednorázově
- 2 predikát se nechová „přirozeně“. Pokud máme dva seznamy ve tvaru  $(a \dots a (b \dots$ , pak je „okamžitě jasné“, že výsledek pro ně by měl být #f, ale předchozí procedura je oba nejprve celé linearizuje

### Odstraníme problém č. 2:

- vstupní seznamy budeme linearizovat do proudů (to jest výsledkem linearizace seznamu bude proud atomů)
- okamžitě budeme mít k dispozici „nejlevější atom“
- ostatní prvky linearizovaného seznamu se budou hledat až když přistoupíme k dalšímu prvku proudů
- vytvoříme predikát na test shody dvou konečných proudů

## Zvýšení výpočetní efektivity pomocí proudů

Každý příslib je roven pouze sám sobě:

```
(define d (delay 1))  
(equal? d d)            $\implies$  #t  
(equal? (delay 1) (delay 1))  $\implies$  #f
```

tím pádem:

```
(equal? (stream 'a 'b 'c) (stream 'a 'b 'c))  $\implies$  #f
```

proto zavádíme predikát shodnosti dvou proudů:

```
(define stream-equal?  
  (lambda (s1 s2)  
    (or (and (null? s1) (null? s2))  
        (and (equal? (stream-car s1)  
                     (stream-car s2))  
              (stream-equal? (stream-cdr s1)  
                              (stream-cdr s2))))))
```



## Zvýšení výpočetní efektivity pomocí proudů

;; téměř dokonalá verze *equal-fringe?*

```
(define equal-fringe?  
  (lambda (s1 s2)
```

;; pomocná definice: *linearizace seznamu*

```
(define flatten  
  (lambda (l)  
    (cond ((null? l) '())  
          ((list? (car l))  
           (stream-append2 (flatten (car l))  
                             (flatten (cdr l))))  
          (else (cons-stream (car l)  
                               (flatten (cdr l)))))))
```

;; jsou lineární seznamy totožné?

```
(stream-equal? (flatten s1) (flatten s2)))
```

## Zvýšení výpočetní efektivity pomocí proudů

### Předchozí řešení má jeden malý problém

Příklad, na kterém náš predikát selhává

```
(define a '(a)) (set-cdr! a a)
(define b '((b))) (set-cdr! b b)
(equal-fringe? a b)  $\implies$   $\infty$  (cyklí)
```

**Odstranění problému** (rozmyslete si proč):

místo procedury `stream-append2` vytvoříme *makro*

```
(define-macro stream-append2
  (lambda (s1 s2)
    '(let proc ((s ,s1))
      (if (stream-null? s)
          ,s2
          (cons-stream (stream-car s)
                        (proc (stream-cdr s)))))))
```

nový pohled: **makro** = „procedura s líně vyhodnocenými argumenty“

## Vyhodnocovací proces s vyrovnávací pamětí

Modelový program

```
(define fib
  (lambda (n)
    (if (<= n 2)
        1
        (+ (fib (- n 1))
            (fib (- n 2)))))))
```

Výhody a nevýhody předchozího kódu:

- **výhoda:** je čistě napsaný (vznikl přepisem definice fib. čísla)
- **nevýhoda:** je neefektivní dochází ke zbytečnému opakování výpočtů

Otázka:

*Jak zachovat čitelnost kódu, ale zvýšit efektivitu výpočetního procesu?*

## Vyhodnocovací proces s vyrovnávací pamětí

;; iterativní verze procedury

;; představuje zrychlení na úkor čitelnosti kódu

```
(define fib
  (lambda (n)
    (let iter ((a 1) (b 1) (n n))
      (if (<= n 1)
          a
          (iter b (+ a b) (- n 1))))))
```

### Lepší řešení:

- zachováme původní kód procedury
- proceduru zabalíme do další procedury (**memoize**), která bude mít svou vnitřní vyrovnávací paměť do které bude ukládat výsledky aplikace výchozí procedury
- odstraníme tak *problém opakovaného provádění stejných výpočtů*

Vytvoříme procedury `empty-assoc` a `assoc!` pro správu paměti

*;; prázdná paměť*

```
(define empty-assoc  
  (lambda () (cons (cons #f #f) '())))
```

*;; destruktivně zařad' nový záznam/modifikuj existující*

```
(define assoc!  
  (lambda (assoc key val)  
    (let iter ((as assoc))  
      (cond ((null? as)  
              (set-cdr! assoc  
                          (cons (cons key val)  
                                (cdr assoc))))  
            ((equal? (caar as) key)  
              (set-cdr! (car as) val))  
            (else (iter (cdr as)))))))
```

Příklad zařazení nových záznamů:

```
(define a (empty-assoc))
```

```
a  $\Rightarrow$  ((#f . #f))
```

```
(assoc! a 'ahoj 10)
```

```
a  $\Rightarrow$  ((#f . #f) (ahoj . 10))
```

```
(assoc! a 'blah 20)
```

```
a  $\Rightarrow$  ((#f . #f) (blah . 20) (ahoj . 10))
```

```
(assoc! a 'ahoj 30)
```

```
a  $\Rightarrow$  ((#f . #f) (blah . 20) (ahoj . 30))
```

Vyhledávání lze provést pomocí klasického `assoc`

```
(assoc 'blah a)  $\Rightarrow$  (blah . 20)
```

```
(assoc #f a)  $\Rightarrow$  (#f . #f)
```

Poznámka: pár `(#f . #f)` je vždy přítomen kvůli mutaci

## Vyhodnocovací proces s vyrovnávací pamětí

- procedura `memoize` vytvoří obálku nad danou procedurou (provede memoizaci dané procedury)
- každá memoizovaná procedura má *vlastní paměť pro úschovu výsledků*
- při volání memoizované procedury s již *dříve použitými argumenty* je výsledná hodnota *vyhledána v paměti*

```
(define memoize
  (lambda (f)
    (let ((memory (empty-assoc)))
      (lambda called-with-args
        (let ((found (assoc called-with-args memory)))
          (if found
              (cdr found)
              (let ((result (apply f called-with-args)))
                (assoc! memory called-with-args result)
                result))))))))))
```

;; Fibonacciho čísla urychlená pomocí *memoize*

```
(define fib
  (memoize
    (lambda (n)
      (if (<= n 2)
          1
          (+ (fib (- n 1))
              (fib (- n 2)))))))
```

**POZOR:** následující by nefungovalo:

```
(define fib (lambda (n) ... původní pomalý fib))
(define fast-fib (memoize fib))
(fast-fib 32) bude ve skutečnosti pomalý
```

- ve *fast-fib* se rekurzivně volá původní procedura bez cache
- při *(fast-fib 32)* je zapamatován pouze výsledek pro 32
- nevede ke kýženému zrychlení výpočtu



Procedury s keší se chovají jinak než běžné procedury  
v případě použití vedlejšího efektu

```
(let ((ctr 0))  
  (define modify  
    (lambda (x)  
      (set! ctr (+ 1 ctr))))  
  (modify #f)  
  (modify #f)  
  ctr)  $\implies$  2
```

```
(let ((ctr 0))  
  (define modify  
    (memoize  
      (lambda (x)  
        (set! ctr (+ 1 ctr)))))  
  (modify #f)  
  (modify #f)  
  ctr)  $\implies$  1
```

*;; vylepšená verze: při přeplnění paměti se paměť vysype*

```
(define make-memoize
  (lambda (limit)
    (lambda (f)
      (let ((memory (empty-assoc)) (memory-size 0))
        (lambda (called-with-args)
          (let ((found (assoc called-with-args memory)))
            (if found
                (cdr found)
                (let ((result (apply f called-with-args)))
                  (if (and (not (null? limit))
                          (> memory-size (car limit)))
                      (begin
                        (set! memory-size 0)
                        (set! memory (empty-assoc)))
                      (assoc! memory called-with-args result)
                      (set! memory-size (+ memory-size 1))
                      result))))))))))
```

;; urychlené *fib* s pamětí o pěti buňkách

```
(define fib
  ((make-memoize 5)
   (lambda (n)
     (if (<= n 2)
         1
         (+ (fib (- n 1))
            (fib (- n 2))))))))
```

Srovnání počtu aktivací vnitřní procedury při dané velikosti paměti  
během výpočtu (*fib* 32)

velikost paměti	50	20	15	10	5	2	1
počet aktivací	32	242	271	1709	6985	75183	287127

## Makro pro vytváření procedur s keší

*;; globální konstanta udávající velikost paměti*

```
(define *max-memory* 1000)
```

*;; makro kappa*

```
(define-macro kappa  
  (lambda (args . body)  
    ‘((make-memoize *max-memory*)  
      (lambda ,args ,@body))))
```

*;; příklad použití:*

```
(define fib  
  (kappa (n)  
    (if (<= n 2)  
        1  
        (+ (fib (- n 1))  
           (fib (- n 2))))))
```