Paradigmata programování 1

Akumulace

Vilém Vychodil

Katedra informatiky, PřF, UP Olomouc

Přednáška 7

Přednáška 7: Přehled

1. Akumulace pomocí `foldr`
   - metody akumulace obcházející problémy s `apply`,
   - `foldr` jako prostředek abstrakce,
   - efektivní implementace vybraných procedur,
   - zobecňování procedur dvou argumentů na procedury libovolně mnoha argumentů,
   - monoidální operace a vliv na akumulaci.
2. Akumulace pomocí `foldl`
   - varianty akumulace pracující „zleva“,
   - implementace variant `foldl` pomocí `foldr`.
3. Motivace pro další přednášku:
   - výpočet faktoriálu,
   - výpočet prvků Fibonacciho posloupnosti.

Opakování: Zpracování seznamů

Mapování

- procedura `map`
- konstruuje nový seznam modifikací prvků výchozího seznamu (více seznamů)
- modifikace prováděna dodanou procedurou

Explicitní aplikace

- procedura `apply`
- aplikuje danou proceduru se seznamem argumentů
- využití: agregace (*akumulace*) prvků seznamu do (jedné) hodnoty

Filtrace

- procedura `filter`, odvozená pomocí `apply` a `append`
- konstruuje nový seznam obsahující pouze prvky splňující vlastnost
- vlastnost je vyjádřena předanou procedurou

Akumulace a nevýhody `apply`

Co je akumulace?

- vytvoření jedné hodnoty (elementu) pomocí více hodnot (elementů) ze seznamu
- *procedura* – *postupně* akumuluje prvky seznamu do hodnoty

Akumulace pomocí `apply`:

- výhody: snadné použití (už známe)
- nevýhody: (obvykle) nutné použít procedury s libovolně mnoha argumenty
  - `(apply + ···`, `(apply append ···`, `(apply map proc ···`

Nový přístup k akumulaci:

- procedura dvou argumentů, postupné aplikace
- např. `(+ 1 (+ 2 (+ 3 4)))` místo `(+ 1 2 3 4)`

Příklad (Motivace pro akumulaci)

`(1 2 3 4)` seznam hodnot

```
(cons 1 (cons 2 (cons 3 (cons 4 '()))))  ⟹  (1 2 3 4)
```

```
(+ 1 (+ 2 (+ 3 4)))
(+ 1 (+ 2 (+ 3 (+ 4 0))))  ⟵ využití neutrálního prvku
```

```
(* 1 (* 2 (* 3 (* 4 1))))
```

```
(list 1 (list 2 (list 3 (list 4 'blah))))
```

Společné a rozdílné rysy:

- stejný styl nabalování (vnořené seznamy)
- různé aplikované procedury: `cons`, `+`, `*`, `list`
- různý způsob terminace v nejvnitřnější aplikaci: `'()`, `0`, `1`, `'blah`

Příklad (Akumulace s procedurou dvou argumentů)

```
(define y+1
  (lambda (x y)
    (+ y 1)))
```

```
(y+1 'a 0)                                  ⟼  1
(y+1 'a (y+1 'b 0))                         ⟼  2
(y+1 'a (y+1 'b (y+1 'c 0)))                ⟼  3
(y+1 'a (y+1 'b (y+1 'c (y+1 'd 0))))       ⟼  4
```

Schematický průběh výpočtu:

```
(y+1 'a (y+1 'b (y+1 'c (y+1 'd 0))))
(y+1 'a (y+1 'b (y+1 'c 1)))
(y+1 'a (y+1 'b 2))
(y+1 'a 3)
4
```

Společný tvar výrazů při akumulaci

Příklady

```
(cons 1 (cons 2 (cons 3 (cons 4 '()))))
(+ 1 (+ 2 (+ 3 (+ 4 0))))
(* 1 (* 2 (* 3 (* 4 1))))
(y+1 1 (y+1 2 (y+1 3 (y+1 4 0))))
```

mají společný tvar:

(⟨*procedura*⟩ 1 (⟨*procedura*⟩ 2 (⟨*procedura*⟩ 3 (⟨*procedura*⟩ 4 ⟨*terminátor*⟩)))),

kde:

- ⟨*procedura*⟩ je procedura (aspoň) dvou argumentů,
- ⟨*terminátor*⟩ je libovolný element.

Definice (procedura `foldr` – *angl. fold right*)

Procedura `foldr` se používá se třemi argumenty ve tvaru:

(`foldr` $\langle proc \rangle$ $\langle term \rangle$ $\langle seznam \rangle$), kde

- argument $\langle proc \rangle$ je procedura dvou argumentů,
- argument $\langle term \rangle$ je libovolný element zvaný terminátor,
- argument $\langle seznam \rangle$ její aplikace je ekvivalentní provedení:

$$(\langle proc \rangle \ \langle e_1 \rangle \ (\langle proc \rangle \ \langle e_2 \rangle \ (\langle proc \rangle \cdots (\langle proc \rangle \ \langle e_n \rangle \ \langle term \rangle) \cdots)))$$

pro $\langle seznam \rangle$ ve tvaru $(\langle e_1 \rangle \ \langle e_2 \rangle \cdots \langle e_n \rangle)$.

- v R5RS není přítomna jako primitivní procedura; v našem interpretu ano
- na `foldr` se lze dívat jako na prostředek abstrakce

Příklad (Předchozí příklady pomocí `foldr`)

Předchozí příklady:

```
(cons 1 (cons 2 (cons 3 (cons 4 '()))))  ⟼  (1 2 3 4)
(+ 1 (+ 2 (+ 3 (+ 4 0))))               ⟼  10
(* 1 (* 2 (* 3 (* 4 1))))               ⟼  24
(y+1 1 (y+1 2 (y+1 3 (y+1 4 0))))       ⟼  4
```

Zobecnění pomocí `foldr`:

```
(define s '(1 2 3 4))
(foldr cons '() s)      ⟼  (1 2 3 4)
(foldr + 0 s)           ⟼  10
(foldr * 1 s)           ⟼  24
(foldr y+1 0 s)         ⟼  4
  ⋮
```

Příklad (Další ukázky použití `foldr`)

Vliv terminátoru na tvar konstruovaného seznamu:

```
(define s '(1 2 3 4))
```

```
(foldr cons 'base s)    ⟹  (1 2 3 4 . base)
(foldr cons '() s)      ⟹  (1 2 3 4)
(foldr list 'base s)    ⟹  (1 (2 (3 (4 base))))
(foldr list '() s)      ⟹  (1 (2 (3 (4 ()))))
```

Mezní případ pro prázdný seznam:

```
(foldr list '() '())    ⟹  ()
(foldr list 'base '())  ⟹  base
(foldr list 666 '())    ⟹  666
```

Poznámky o činnosti `foldr`

(`foldr` $\langle proc \rangle$ $\langle term \rangle$ $\langle seznam \rangle$) =
($\langle proc \rangle$ $\langle e_1 \rangle$ ($\langle proc \rangle$ $\langle e_2 \rangle$ ($\langle proc \rangle \cdots$ ($\langle proc \rangle$ $\langle e_n \rangle$ $\langle term \rangle$) $\cdots$)))

Význam argumentů `foldr`:

1. terminátor $\langle term \rangle$
   - hodnota, kterou `foldr` vrací pro prázdný seznam
   - výsledek zabalení prázdného seznamu pomocí `foldr` je $\langle term \rangle$
2. argumenty akumulační procedury $\langle proc \rangle$:
   - *první argument* = průběžný prvek x seznamu,
   - *druhý argument* = výsledek zabalení hodnot následující v seznamu za x

Kdy použít `foldr`?

- když je potřeba zpracovat prvky seznamu jeden po druhém,
- když je výhodné postupně zpracovávat prvky „od konce seznamu“.

Intermezzo: Připomenutí pojmů z teorie složitosti

Časová složitost algoritmu v nejhorším případě: zobrazení $T\colon \mathbb{N} \to \mathbb{N}$

- $T(n) =$ počet elementárních kroků potřebných pro provedení výpočtu podle daného algoritmu pro vstupní data délky n a to v nejhorším možném případě (největší trvání délky výpočtu, ze všech možných vstupů délky n)
- v naší terminologii: „provedení výpočtu algoritmu“ = „aplikace procedury“

Co je (pro nás) elementárním krokem?

- při práci se seznamy: provedení `cons`, `car`, `cdr` (mírné zjednodušení),
- vytvoření lokálního prostředí při aplikaci procedury,
- aritmetické operace: `+`, `*`, ... (velké zjednodušení)
  - racionální čísla ve Scheme – operace nad nimi neprobíhají v konstantním čase
  - toto zjednodušení přijímáme kvůli snazší analýze
  - je bez újmy, v PP se neorientujeme se na „numerické výpočty“

viz kurs *Algoritmická matematika*

Příklad (Délka seznamu efektivně: procedura `length`)

```
(define length
  (lambda (l)
    (apply +
           (map (lambda (x) 1)
                l))))
```

$\Longrightarrow$

```
(define length
  (lambda (l)
    (foldr (lambda (x y)
             (+ 1 y))
           0 l)))
```

Časová složitost:

1. $T(n) = 3n$ (nebo $4n$, záleží na míře zjednodušení)
2. $T(n) = n$ (nebo $2n$, záleží na míře zjednodušení)

Složitost jsme stanovili za předpokladu, že `+`, `map` a `foldr` pracují v lineárním čase v závislosti na počtu sčítanců / délce seznamu (zjednodušení, ale ne velké).

Diskuse: je `+` stejně náročné jako `cons`?

Příklad (Zřetězení dvou seznamů efektivně: procedura append2)

```
(define append2
  (lambda (l1 l2)
    (let ((len1 (length l1))
          (len2 (length l2)))
      (build-list (+ len1 len2)
                  (lambda (i)
                    (if (< i len1)
                        (list-ref l1 i)
                        (list-ref l2 (- i len1)))))))))
```

$\Longrightarrow$

```
(define append2
  (lambda (l1 l2)
    (foldr cons l2 l1)))
```

Časová složitost:

1. $T(n) = n(n+3)$ protože $\frac{n(n+3)+m(m+3)}{2}$ kroků pro seznamy délek m a n,
2. $T(n) = 2n$, kde n je délka *prvního seznamu*, výrazně efektivnější (**!!**).

Stejné zjednodušující předpoklady jako pro předchozí případ.

Příklad (Zřetězení seznamů: procedura append)

S využitím předchozího:

```
(define append2
  (lambda (l1 l2)
    (foldr cons l2 l1)))
```

Obecný append s libovolně mnoha argumenty:

```
(define append
  (lambda lists
    (foldr append2 '() lists)))
```

- elegantní použití foldr,
- časová složitost: $T(n) = n$, kde n je délka zřetězení všech seznamů
- mírné zhoršení v případě dvou seznamů
  - daň za eleganci (navíc není nijak dramatické),
  - lze napravit dodatečným rozlišením případů v těle append.

Příklad (Mapování přes jeden seznam efektivně: procedura map1)

```
(define map1
  (lambda (f l)
    (build-list
      (length l)
      (lambda (i)
        (f (list-ref l i))))))
```

$\Longrightarrow$

```
(define map1
  (lambda (f l)
    (foldr (lambda (x y)
             (cons (f x) y))
           '()
           l)))
```

Časová složitost:

1. $T(n) = \frac{n(n+3+2t_f)}{2}$,
2. $T(n) = n(2+t_f)$,

kde člen t_f vyjadřuje časovou složitost aplikace `f` na prvek seznamu (pro stanovení celkové složitosti nutno počítat se složitostí aplikace `f`).

Zde *záleží na*: struktuře seznamu `l` a složitosti `f`.

Příklad (Filtrování efektivně: procedura `filter`)

```
(define filter
  (lambda (f l)
    (apply append
           (map (lambda (x)
                  (if (f x)
                      (list x)
                      '()))
                l))))
```

$\Longrightarrow$

```
(define filter
  (lambda (f l)
    (foldr (lambda (x y)
             (if (f x)
                 (cons x y)
                 y))
           '()
           l)))
```

Časová složitost:

1. $T(n) = n + n(3 + t_f)$ (lineární),
2. $T(n) = n(2 + t_f)$ (lineární, ale efektivnější),

kde člen t_f vyjadřuje časovou složitost aplikace predikátu `f` na prvek seznamu.

Definice (procedura `foldr` pro obecně mnoho seznamů)

Proceduru `foldr` lze použít s více seznamy ve tvaru:

$$(\texttt{foldr}\ \langle proc\rangle\ \langle term\rangle\ \langle seznam_1\rangle\ \langle seznam_2\rangle \cdots \langle seznam_k\rangle),$$

- argument $\langle proc\rangle$ je procedura $k+1$ argumentů,
- argument $\langle term\rangle$ je libovolný element zvaný terminátor,
- každý $\langle seznam_i\rangle$ je seznam elementů stejné délky.

Její aplikace je ekvivalentní provedení:

$$\begin{array}{l}(\langle proc\rangle\ \langle e_{1,1}\rangle\ \langle e_{2,1}\rangle \cdots \langle e_{k,1}\rangle \\ \quad (\langle proc\rangle\ \langle e_{1,2}\rangle\ \langle e_{2,2}\rangle \cdots \langle e_{k,2}\rangle \\ \quad\quad (\langle proc\rangle \cdots \\ \quad\quad\quad (\langle proc\rangle\ \langle e_{1,n}\rangle\ \langle e_{2,n}\rangle \cdots \langle e_{k,n}\rangle\ \langle term\rangle) \cdots)))\end{array}$$

pro každý $\langle seznam_i\rangle$ ve tvaru $(\langle e_{i,1}\rangle\ \langle e_{i,1}\rangle \cdots \langle e_{i,1}\rangle)$.

Příklad (Obecná verze `map` pomocí `foldr`, pomocná procedura)

Cíl: Vytvořit obecnou verzi `map` pomocí `foldr`.

Přináší s sebou:

- explicitní aplikaci `foldr`
- související problém: separace posledního prvku seznamu.

Pomocná procedura `separate-last-argument`:

```
(define separate-last-argument
  (lambda (l)
    (foldr (lambda (x y)
             (if (not y)
                 (cons '() x)
                 (cons (cons x (car y)) (cdr y))))
           #f
           l)))
```

Příklad (Obecná verze `map` pomocí `foldr`)

Výsledné řešení:

```
(define map
  (lambda (f . lists)
    (apply foldr
           (lambda args
             (let ((separation (separate-last-argument args)))
               (cons (apply f (car separation))
                     (cdr separation))))
           '()
           lists)))
```

- $T(k,n) = kn + n(3 + t_f + k) = n(3 + 2k + t_f)$
- řádově nejlepší řešení, prakticky ale ukážeme efektivnější (další přednášky)
- neefektivita spočívá v nasazení `separate-last-argument`

Intermezzo: Monoidální operace

Definice (monoidální zobrazení)

Zobrazení $f : A \times A \to A$ nazveme **monoidální**, pokud jsou splněny tyto podmínky:

1. existuje $e \in A$ takový, že pro každý $a \in A$ platí $f(a, e) = f(e, a) = a$,
2. pro každé $a, b, c \in A$ platí, že $f(a, f(b, c)) = f(f(a, b), c)$.

Terminologie:

- zobrazení $f : A \times A \to A$ je **binární operace na množině** A
- výsledek operace $\circ : A \times A \to A$ pro hodnoty $a, b \in A$ značíme $a \circ b$ místo $\circ(a, b)$

Důsledek – **monoidální operace** (odvozený pojem)

Binární operace $\circ$ na A je monoidální operace, pokud

1. $\circ$ je asociativní ($a \circ (b \circ c) = (a \circ b) \circ c$ pro každé $a, b, c \in A$),
2. $\circ$ má neutrální prvek (existuje $e \in A$ tak, že $a \circ e = e \circ a = a$ pro každý $a \in A$).

Příklad (Příklady monoidálních operací)

Příklady z matematiky:

- $+$ (sčítání čísel) na $\mathbb{R}$ je monoidální (neutrální prvek je 0)
- $\cdot$ (násobení čísel) na $\mathbb{R}$ je monoidální (neutrální prvek je 1)
- $\cap$ (průnik množin) na 2^X je monoidální (neutrální prvek je X)
- $\cup$ (sjednocení množin) na 2^X je monoidální (neutrální prvek je $\emptyset$)
- ... mnoho dalších (např. sčítání polynomů / matic, apod.)

Příklady z informatiky:

- všechny předchozí, ... (omezené na čísla / množiny reprezentovatelné v počítači)
- append na množině všech seznamů je monoidální
  (neutrální prvek je prázdný seznam)
- ... mnoho dalších (viz kurs *Formální jazyky a automaty*)

Monoidální operace a `foldr`

Proč jsou monoidální operace důležité?

- důsledek asociativity: *závorkování nemá význam*
- důsledek neutralita: operace mají přirozený *význam i bez argumentů*

Rozšíření monoidální operace $\circledcirc$ na libovolně mnoho argumentů:

$$a_1 \circledcirc a_2 \circledcirc \cdots \circledcirc a_n$$
$$a_1 \circledcirc a_2 \circledcirc \cdots \circledcirc a_n = a_1 \circledcirc a_2 \circledcirc \cdots \circledcirc a_n \circledcirc e$$
$$a_1 \circledcirc a_2 \circledcirc \cdots \circledcirc a_n = (a_1 \circledcirc (a_2 \circledcirc \cdots (a_n \circledcirc e) \cdots))$$

Poslední uvedený koresponduje se

`(foldr` $\circledcirc$ e $\langle seznam \rangle$`)`,

kde $\langle seznam \rangle$ obsahuje prvky $a_1, \ldots, a_n$.

Příklad (Příklady rozšíření operací na libovolné argumenty)

Rozšíření sčítání dvou argumentů:

```
(define add2 (lambda (x y) (+ x y)))

(define add
  (lambda args
    (foldr add2 0 args)))
```

Rozšíření sjednocení množin:

```
(define union2
  (set-operation (lambda (x A B)   ⟵ předchozí přednáška
                   (or (in? x A) (in? x B)))))

(define union
  (lambda sets
    (foldr union2 '() sets)))
```

Příklad (Problém s neexistencí neutrálního prvku I)

Pokud je operace asociativní, ale nemá neutrální prvek:

1. zobecníme operaci pro ≥ 1 argumentů (např. primitivní procedura `min`),
2. dodáme nový neutrální prvek „uměle“.

První typ řešení:

```
(define min2
  (lambda (x y)
    (if (<= x y) x y)))

(define min
  (lambda numbers
    (foldr min2 (car numbers) (cdr numbers))))
```

Příklad (Problém s neexistencí neutrálního prvku II)

Druhý typ řešení:

```
(define +infty '+infty)
```
⟵ přidaný neutrální prvek

```
(define <=
  (let ((<= <=))
    (lambda (x y)
      (or (equal? y +infty)
          (and (not (equal? x +infty))
               (<= x y))))))

(define min
  (lambda numbers
    (foldr min2 +infty numbers)))
```

Procedury `genuine-foldl` a `foldl`

Zabalení „zleva doprava“ pomocí `foldr`:

$$(\texttt{foldr}\ \langle proc\rangle\ \langle term\rangle\ \langle seznam\rangle) = \\ (\langle proc\rangle\ \langle e_1\rangle\ (\langle proc\rangle\ \langle e_2\rangle\ (\langle proc\rangle \cdots (\langle proc\rangle\ \langle e_n\rangle\ \langle term\rangle) \cdots)))$$

Dvě možnosti zabalení „zprava doleva“:

Procedura `foldl`:

$$(\texttt{foldl}\ \langle proc\rangle\ \langle term\rangle\ \langle seznam\rangle) = \\ (\langle proc\rangle\ \langle e_n\rangle\ (\langle proc\rangle\ \langle e_{n-1}\rangle\ (\langle proc\rangle \cdots (\langle proc\rangle\ \langle e_1\rangle\ \langle term\rangle) \cdots)))$$

Procedura `genuine-foldl`:

$$(\texttt{genuine-foldl}\ \langle proc\rangle\ \langle term\rangle\ \langle seznam\rangle) = \\ (\langle proc\rangle\ (\langle proc\rangle \cdots (\langle proc\rangle\ (\langle proc\rangle\ \langle term\rangle\ \langle e_1\rangle)\ \langle e_2\rangle) \cdots)\ \langle e_n\rangle)$$

Pro monoidální operace platí: `foldr` $=$ `genuine-foldl` (**!!**)

Příklad (Rozdíly mezi `foldr`, `foldl` a `genuine-foldl`)

Rozdíly v akumulaci:

```
(define proc
  (lambda (x y)
    (list #f x y)))

(define s '(a b c d))
```

```
(foldr proc 'x s)          ⟼  (#f a (#f b (#f c (#f d x))))
(foldl proc 'x s)          ⟼  (#f d (#f c (#f b (#f a x))))
(genuine-foldl proc 'x s)  ⟼  (#f (#f (#f (#f x a) b) c) d)
```

Použití pro obrácení seznamu:

```
(define reverse
  (lambda (l)
    (foldl cons '() l)))
```

Příklad (Implementace `foldl` pomocí `foldr`)

(`foldl` $\langle proc \rangle$ $\langle term \rangle$ $\langle seznam \rangle$) =
(($\langle proc \rangle$ $\langle e_n \rangle$ (($\langle proc \rangle$ $\langle e_{n-1} \rangle$ (($\langle proc \rangle \cdots$ (($\langle proc \rangle$ $\langle e_1 \rangle$ $\langle term \rangle$)$\cdots$)))

```
(define foldl
  (lambda (f term l)
    (foldr f term (reverse l))))
```

Příklad (Implementace `genuine-foldl` pomocí `foldr`)

(`genuine-foldl` $\langle proc \rangle$ $\langle term \rangle$ $\langle seznam \rangle$) =
($\langle proc \rangle$ ($\langle proc \rangle \cdots$ (($\langle proc \rangle$ ($\langle proc \rangle$ $\langle term \rangle$ $\langle e_1 \rangle$) $\langle e_2 \rangle$)$\cdots$) $\langle e_n \rangle$)

```
(define genuine-foldl
  (lambda (f term l)
    (foldr (lambda (x y) (f y x))
           term
           (reverse l))))
```

Příklad (Složení funkcí: procedura compose)

Připomeňme: Složení zobrazení $f\colon X \to X$ a $g\colon X \to X$ (v tomto pořadí) je zobrazení $(f \circ g)\colon X \to X$ definované

$$(f \circ g)(x) = g\big(f(x)\big) \quad \text{pro každé } x \in X.$$

Vlastnosti operace $\circ$ skládání:

$$f \circ (g \circ h) = (f \circ g) \circ h, \qquad \text{asociativita,}$$
$$\iota \circ f = f \circ \iota = f, \qquad \text{neutralita vzhledem k } \iota(x) = x.$$

```
(define compose
  (lambda functions
    (foldl (lambda (f g)
             (lambda (x) (f (g x))))
           (lambda (x) x)
           functions)))
```

Jak vypadají prostředí?

Poznámka: rozšíření na libovolně mnoho seznamů

(foldr $\langle proc \rangle$ $\langle term \rangle$ $\langle seznam_1 \rangle$ $\langle seznam_2 \rangle \cdots \langle seznam_k \rangle$)
(genuine-foldr $\langle proc \rangle$ $\langle term \rangle$ $\langle seznam_1 \rangle$ $\langle seznam_2 \rangle \cdots \langle seznam_k \rangle$)

Implementace:

```
(define genuine-foldl
  (lambda (f term . lists)
    (apply foldr
           (lambda args
             (apply f (reverse args)))
           term
           (map reverse lists))))

(define foldl
  (lambda (f term . lists)
    (apply foldr f term (map reverse lists))))
```

Příklad (Výpočet faktoriálu pomocí `foldr`)

Faktoriál:

$$n! = \underbrace{1 \cdot 2 \cdot \cdots \cdot n}_{n \text{ činitelů}} \quad \text{pro } n \geq 0$$

Poznámka: mezní případ je $0! = 1$.

Lze počítat pomocí `foldr`:

```
(define fac
  (lambda (n)
    (foldr * 1 (build-list n (lambda (x) (+ x 1))))))
```

- lze rovněž pomocí `apply`,
- neefektivní, protože konstruujeme seznam činitelů,
- řešení není programátorsky elegantní.

Příklad (Výpočet prvků Fibonacciho posloupnosti pomocí `foldr`)

Fibonacciho posloupnost:

n:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15 ...
$F(n)$:	0	1	1	2	3	5	8	13	21	34	55	89	144	233	377	610 ...

Lze počítat pomocí `foldr`:

```
(define fib
  (lambda (n)
    (cadr
     (foldr (lambda (x last)
              (list (apply + last) (car last)))
            '(1 0)
            (build-list n (lambda (x) #f))))))
```

Příští přednáška: *uvidíme efektivní a elegantní řešení*