Paradigmata programování 1

Explicitní aplikace a vyhodnocování

Vilém Vychodil

Katedra informatiky, PřF, UP Olomouc

Přednáška 6

Přednáška 6: Přehled

1. Explicitní aplikace procedur:
   - Apply je dostupné prostřednictvím procedury `apply`,
   - provádění operací se seznamy využívající `apply`,
   - uživatelsky definované procedury s nepovinnými a libovolnými argumenty.
2. Explicitní vyhodnocování elementů:
   - prostředí jako element prvního řádu,
   - Eval (součást REPL) je dostupné prostřednictvím procedury `eval`,
   - data = program se vším všudy,
   - využití a záludnosti `eval`.
3. Praktické příklady použití:
   - množiny reprezentované seznamy hodnot (bez duplicit),
   - binární relace (mezi množinami) reprezentované jako seznamy párů.

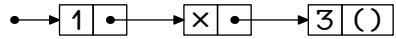
Opakování

Páry a seznamy

- **tečkový pár**: agreguje dvě hodnoty (`car`, `cdr`, `cons`)
- **prázdný seznam**: `()` speciální element jazyka
- **seznamy**: definované (rekurzivně) každý element, který je buď
  - *prázdný seznam*, nebo
  - *pár* vytvořený aplikací `cons` na *libovolný element* a *seznam* (v tomto pořadí).

```
(cons 1 (cons 'x (cons 3 '())))
```

$\Longmapsto$



Operace se seznamy (zatím známe):

- konstruktory: `cons`, `list`, `build-list` (procedura vyššího řádu),
- selektory: `car`, `cdr`, `list-ref`
- další operace: `append`, `reverse`
- mapování: `map` (procedura vyššího řádu)

Aplikace (uživatelsky definovaných procedur), Př. 2–3

Mějme proceduru F ve tvaru

$$\langle (\langle param_1 \rangle \ \langle param_2 \rangle \cdots \langle param_n \rangle); \langle výraz_1 \rangle, \langle výraz_2 \rangle, \ldots, \langle výraz_m \rangle; \mathcal{P} \rangle$$

$\mathrm{Apply}[F; E_1, \ldots, E_m]$ = **aplikace procedury F na argumenty $E_1, \ldots, E_m$**
je hodnota (element), která je získána následovně:

- pokud $m \neq n$, končíme chybou (špatný počet argumentů), jinak:
- je vytvořeno nové lokální prostředí $\mathcal{P}_l$
- předek prostředí $\mathcal{P}_l$ je nastaven na $\mathcal{P}$
- v $\mathcal{P}_l$ se zavedou vazby $\langle param_n \rangle \mapsto E_i$
- v $\mathcal{P}_l$ se postupně vyhodnotí $\langle výraz_1 \rangle, \langle výraz_2 \rangle, \ldots, \langle výraz_m \rangle$
- $\mathrm{Apply}[F; E_1, \ldots, E_m]$ je potom výsledkem vyhodnocení $\langle výraz_m \rangle$

Implicitní × explicitní aplikace

Rozeznáváme dva typy aplikací

1. **implicitní** – doposud
   - aplikace procedur, která je důsledkem vyhodnocování seznamů
   - probíhá *implicitně* (je zahrnut, ale není přímo vyjádřen)
2. **explicitní** – dnes nové
   - aplikace procedur, která probíhá na naši žádost
   - probíhá *explicitně* (žádost o aplikaci je přímo vyjádřená)

Jak a kdy použít explicitní aplikaci?

- Lze použít pomocí procedury `apply`, uvidíme dále.
- Používá se v případě, kdy máme seznam argumentů k dispozici (jako data).

Příklad (Motivační příklad pro `apply`)

Sečtení všech čísel v seznamu:

`s` $\longmapsto$ seznam čísel

`(+ (car s) (cadr s) (caddr s) ⋯)` $\longleftarrow$ co když neznáme délku?

`(+ s)` $\longmapsto$ „CHYBA: argument pro + není číslo“

Řešení:

```
(apply + s)
```

Význam:

- vyžádáme hodnotu $\mathrm{Apply}[F; E_1, \ldots, E_n]$, kde
  - F je procedura sčítání
  - prvky seznamu navázaného na `s` jsou $E_1, \ldots, E_n$

Definice (primitivní procedura `apply`)

Primitivní procedura `apply` se používá s argumenty ve tvaru:

(`apply` $\langle procedura \rangle$ $\langle arg_1 \rangle$ $\langle arg_2 \rangle \cdots \langle arg_n \rangle$ $\langle seznam \rangle$), kde

- argument $\langle procedura \rangle$ je procedura,
- $\langle arg_1 \rangle, \ldots, \langle arg_n \rangle$ jsou nepovinné argumenty, a mohou být vynechány,
- argument $\langle seznam \rangle$ je seznam a musí být uveden.

Aplikace probíhá následovně:

1. Je sestaven seznam argumentů:
   - $\langle arg_1 \rangle$ je první prvek sestaveného seznamu,
   - $\langle arg_2 \rangle$ je druhý prvek sestaveného seznamu,
   - $\vdots$
   - $\langle arg_n \rangle$ je n-tý prvek sestaveného seznamu,
   - zbylé prvky jsou doplněny ze seznamu $\langle seznam \rangle$.
2. $\langle procedura \rangle$ je aplikována s argumenty ze sestaveného seznamu.

Příklad (Používání `apply`)

```
(apply + '())                          ⟼  0
(apply append '())                     ⟼  ()
(apply append '((1 2 3) () (c d)))     ⟼  (1 2 3 c d)
(apply list '(1 2 3 4))                ⟼  (1 2 3 4)
(apply cons (list 2 3))                ⟼  (2 . 3)
(apply min '(4 1 3 2))                 ⟼  1
```

```
(apply +)                  ⟼  „CHYBA: Chybí seznam hodnot.“
(apply cons '(1 2 3 4))    ⟼  „CHYBA: cons má mít dva argumenty.“
```

```
(apply + 1 2 3 4 '())   ⟼  10
(apply + 1 2 3 '(4))    ⟼  10
(apply + 1 2 '(3 4))    ⟼  10
(apply + 1 '(2 3 4))    ⟼  10
(apply + '(1 2 3 4))    ⟼  10
```

Příklad (Použití `apply`: transpozice matic reprezentovaných seznamy)

Motivace:

```
(apply map list '((a b c) (1 2 3)))  ⟹  ((a 1) (b 2) (c 3))
```

Reprezentace matice a její transpozice:

$$\mathbf{A} = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{pmatrix}, \quad \mathbf{A}^T = \begin{pmatrix} 1 & 5 & 9 \\ 2 & 6 & 10 \\ 3 & 7 & 11 \\ 4 & 8 & 12 \end{pmatrix} \quad \equiv$$

```
((1 5 9)
 (2 6 10)
 (3 7 11)
 (4 8 12))
```

Řešení:

```
(define transpose
  (lambda (matrix)
    (apply map list matrix)))
```

Příklad (Výpočet délky seznamu: `length`)

Pozorování:

```
(define s '(a b c d e f g))
s  ⟼  (a b c d e f g)

(map (lambda (x) 1) s)  ⟼  (1 1 1 1 1 1 1)

(apply + (map (lambda (x) 1) s))  ⟼  7
```

Zobecnění pozorování pro výpočet délky seznamu:

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

- je „mírně neefektivní“, ale není to žádná „katastrofa“
- mezní případ: `(length '())` funguje korektně díky „součtu nula jedniček“ (**!!**)

Filtrace – metoda zpracování seznamu

- procedura `filter`
- argumenty: $\langle funkce \rangle$ jednoho argumentu (podmínka) a $\langle seznam \rangle$
- výsledek aplikace: seznam prvků z $\langle seznam \rangle$ splňujících podmínku

Příklad (Příklady použití `filter`)

```
(filter even? '(1 2 3 4 5 6))              ⟼  (2 4 6)
(filter pair? '(1 (2 . a) 3 (4 . k)))      ⟼  ((2 . a) (4 . k))
(filter (lambda (x)
          (not (pair? x)))
        '(1 (2 . a) 3 (4 . k)))            ⟼  (1 3)
(filter symbol? '(1 a 3 b 4 d))            ⟼  (a b d)
```

Ouha! Procedura `filter` není v jazyku Scheme (standardně) implementována.

Příklad (Jak chápat filtraci?)

```
(define s '(1 3 2 6 1 7 4 8 9 3 4))

(map (lambda (x)
       (if (even? x)
           (list x)
           '()))
     s)  ⟹  (() () (2) (6) () () (4) (8) () () (4))

(apply append
       (map (lambda (x)
              (if (even? x)
                  (list x)
                  '()))
            s))  ⟹  (2 6 4 8 4)
```

Příklad (Implementace procedury `filter`)

Zobecnění předchozího pozorování:

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

Příklad (Procedury `remove` a `member?` vytvořené pomocí filtrace)

Odstranění prvků splňující podmínku:

```
(define remove
  (lambda (f l)
    (filter (lambda (x)
              (not (f x)))
            l)))
```

Test přítomnosti prvku v seznamu:

```
(define member?
  (lambda (elem l)
    (not (null? (filter
                 (lambda (x)
                   (equal? x elem))
                 l)))))
```

Příklad (Nepříliš efektivní implementace `list-ref`)

```
(define list-ref
  (lambda (l index)
    (let ((indices (build-list (length l) (lambda (i) i))))
      (cdar
       (filter (lambda (cell)
                 (= (car cell) index))
               (map cons indices l))))))
```

- opět neefektivní
- konstruuje se „zbytečný“ seznam indexů
- konstruuje se „zbytečný“ seznam párů index-hodnota
- lepší řešení – na dalších přednáškách

Definice (λ-výraz s nepovinnými a libovolnými formálními argumenty)

Každý seznam ve tvaru

(lambda ($\langle param_1 \rangle \cdots \langle param_m \rangle$) $\langle výraz_1 \rangle \cdots \langle výraz_k \rangle$), nebo
(lambda ($\langle param_1 \rangle \cdots \langle param_n \rangle$. $\langle zbytek \rangle$) $\langle výraz_1 \rangle \cdots \langle výraz_k \rangle$), nebo
(lambda $\langle parametry \rangle$ $\langle výraz_1 \rangle \cdots \langle výraz_k \rangle$), kde

- n, k jsou kladná čísla, m je nezáporné číslo,
- $\langle param_1 \rangle, \ldots, \langle param_n \rangle, \langle zbytek \rangle$ jsou vzájemně různé symboly,
- $\langle parametry \rangle$ je symbol,
- $\langle výraz_1 \rangle, \ldots, \langle výraz_k \rangle$ jsou libovolné výrazy tvořící tělo,

se nazývá **λ-výraz**, přitom:

- m, n nazýváme *počet povinných formálních argumentů*,
- $\langle zbytek \rangle$ je formální argument *zastupující seznam nepovinných argumentů*,
- $\langle parametry \rangle$ je formální argument *zastupující seznam všech argumentů*.

Příklad (Porovnání čísel s danou přesností)

```
(define approx=
  (lambda (x y . epsilon)
    (let ((epsilon
           (if (null? epsilon)
               1/1000
               (car epsilon))))
      (<= (abs (- x y)) epsilon))))
```

```
(approx= 5 5.00027)      ⟹  #t
(approx= 5 5.01)         ⟹  #f
(approx= 5 5.01 1/100)   ⟹  #t
```

Příklad (Vytvoření obecného `map` pomocí `map1` a `apply`)

```
(define map
  (lambda (f . lists)
    (let ((len (length (car lists))))
      (build-list len
                  (lambda (index)
                    (apply f
                           (map1 (lambda (l)
                                   (list-ref l index))
                                 lists)))))))
```

```
(map list '(a b c))           ⟼  ((a) (b) (c))
(map cons '(a b c) '(1 2 3))  ⟼  ((a . 1) (b . 2) (c . 3))
```

- není efektivní kvůli použití `list-ref`

Příklad (Procedury zpracovávající libovolný počet argumentů)

Součet čtverců:

```
(define sum2
  (lambda cisla
    (apply + (map (lambda (x) (* x x)) cisla))))
```

```
(sum2)          ⟼  0
(sum2 1)        ⟼  1
(sum2 1 2)      ⟼  5
(sum2 1 2 3)    ⟼  14
```

Užitečné procedury:

```
(define first-arg (lambda list (car list)))
(define second-arg (lambda list (cadr list)))
(define always (lambda (x) (lambda list x)))
  ⋮
```

Explicitní vyhodnocování a `eval`

Eval je k dispozici podobně jako Apply:

- primitivní procedura `eval` – argumentem je element k vyhodnocení
- explicitní vyhodnocení pomocí `eval` probíhá v **globálním prostředí**
- používat s mírou (**!!**)

```
(eval 10)              ⟹  10
(eval '+)              ⟹  „procedura“
(eval '(+ 1 2))        ⟹  3
(eval (list + 1 2))    ⟹  3            ⟵ zajímavé, . . .

(let ((x 10)) (eval '(+ 5 x)))         ⟹  „CHYBA: x nemá vazbu“
(let ((x 10)) (eval (list '+ 5 x)))    ⟹  15

(eval (read))          ⟹  proveď jeden cyklus REPLu
(eval (list * 2 (read)))  ⟹  zakomponuj vstup do výrazu a vyhodnoť
```

Příklad (Predikáty `forall` a `and-proc`)

Řešení pomocí filtrace (pozor, `apply` nelze použít přímo s `and`):

```
(define and-proc
  (lambda args
    (null? (remove (lambda (x) x) args))))

(define forall
  (lambda (f l)
    (apply and-proc (map f l))))
```

Nešťastné řešení pomocí `eval`:

```
(define and-proc
  (lambda args
    (eval (cons 'and args))))
```

`(and-proc '(if #f #t #f))` $\longmapsto$ `#f`

`(and '(if #f #t #f))` $\longmapsto$ `(if #f #t #f)` což je „true“

Prostředí jako element prvního řádu

Omezení předchozího `eval`:

- $\mathrm{Eval}[E, \mathcal{P}]$ versus $\mathrm{Eval}[E, \mathcal{P}_G]$
- je potřeba mít `eval` se dvěma argumenty: **elementem** a **prostředím**
- je potřeba „zhmotnit prostředí“ a pracovat s ním jako s elementem jazyka

Prostředky pro práci s prostředími

- `(the-environment)` – vrací aktuální prostředí (speciální forma)
- `(environment-parent` $\langle prostředí \rangle$`)` – vrací předchůdce $\langle prostředí \rangle$
- `(procedure-environment` $\langle procedura \rangle$`)` – vrací prostředí vzniku procedury
- `(environment->list` $\langle prostředí \rangle$`)` – vrací seznam vazeb v $\langle prostředí \rangle$

Prostředí je *element prvního řádu*. (!!)

Příklad (Explicitní vyhodnocování v lokálních prostředích)

```
(let ((x 10))
  (the-environment))    ⟹   „lokální prostředí, kde x ↦ 10“

(let ((x 10))
  (eval '(* 2 x)
        (the-environment)))   ⟹   20

(eval '(* 2 x)
      (let ((x 10))
        (the-environment)))   ⟹   20

(let ((x 10)
      (y 20))
  (environment->list (the-environment)))
    ⟹   ((x . 10) (y . 20))
```

Příklad (Získání prostředí vzniku procedury a lexikálního předka)

```
(procedure-environment
 (let ((x 10))
   (lambda (y)
     (+ x y))))  ⟹  prostředí vzniku procedury, kde x ↦ 10

(environment->list
 (environment-parent
  (let* ((x 10)
         (y 20))
    (the-environment))))  ⟹  ((x . 10))
```

- `the-environment`, `procedure-environment`, `environment->list` je k dispozici (například) v interpretu Elk (embeddable Scheme)
- tyto procedury a formy nejsou k dispozici všude (**!!**)

Příklad (Proč je použití `eval` nebezpečné?)

- je možné měnit (mutovat) prostředí vzniku procedur, aniž by to bylo patrné
- nebezpečí vzniku chyb

```
(define aux-proc
  (let ()
    (lambda (x)
      (+ x y))))
```

```
(aux-proc 10)  ⟼  „CHYBA: Symbol y nemá vazbu.“
```

```
(eval '(define y 20)
      (procedure-environment aux-proc))
```

```
y              ⟼  „Symbol y nemá vazbu“
(aux-proc 10)  ⟼  30 „zdánlivě podivné“
```

Příklad: Množiny a binární relace

Konečné množiny

- matematické struktury: $A = \{a_1, a_2, \ldots, a_n\}$, kde $n \geq 0$ (pro $n = 0$ je $A = \emptyset$)
- reprezentace ve Scheme – *seznamem hodnot* (bez duplicit)

Kartézský součin množin A a B (v tomto pořadí)

- $A \times B = \{\langle x, y\rangle \mid x \in A \text{ a } y \in B\}$
- pro $A = \{a, b, c\}$, $B = \{1, 2\}$: $A \times B = \{\langle a, 1\rangle, \langle a, 2\rangle, \langle b, 1\rangle, \langle b, 2\rangle, \langle c, 1\rangle, \langle c, 2\rangle\}$
- pro konečné A a B lze $A \times B$ reprezentovat *seznamem párů*

Binární relace (relace mezi dvěma množinami A a B, v tomto pořadí)

- libovolná (tj. jakákoliv) podmnožina $A \times B$
- značení: $R \subseteq A \times B$

Ukážeme: implementace operací s množinami a relacemi ve Scheme

Příklad (Základní konstruktory pro práci s množinami)

```
(define the-empty-set '())

(define card
  (lambda (set)
    (length set)))

(define make-set
  (lambda (prop? universe)
    (filter prop? universe)))

(define cons-set
  (lambda (x set)
    (if (in? x set)          ⟵ test přítomnosti prvku v množině
        set
        (cons x set))))
```

Příklad (Základní predikáty pro práci s množinami)

```
(define in?
  (lambda (x set)
    (not (null? (filter (lambda (y) (equal? x y)) set)))))

(define set?
  (lambda (elem)
    (and (list? elem)
         (forall (lambda (x)
                   (= (occurrences x elem) 1))
                 elem))))

(define occurrences
  (lambda (elem l)
    (length (filter (lambda (x) (equal? x elem)) l))))
```

Příklad (Operace s množinami: průnik a sjednocení)

$A \cup B = \{x \mid x \in A$ nebo $x \in B\}$ (sjednocení)
$A \cap B = \{x \mid x \in A$ a $x \in B\}$ (průnik)

```
(define union
  (lambda (set-A set-B)
    (list->set (append set-A set-B))))
```
← třeba dodělat `list->set`

```
(define intersection
  (lambda (set-A set-B)
    (make-set (lambda (x)
                (and (in? x set-A)
                     (in? x set-B)))
              (union set-A set-B))))
```

Jak zobecnit?

Příklad (Vytváření obecných množinových operací)

```
(define set-operation
  (lambda (prop)
    (lambda (set-A set-B)
      (filter (lambda (x)
                (prop x set-A set-B))
              (list->set (append set-A set-B))))))
(define union
  (set-operation (lambda (x A B)
                   (or (in? x A) (in? x B)))))
(define intersection
  (set-operation (lambda (x A B)
                   (and (in? x A) (in? x B)))))
(define set-minus
  (set-operation (lambda (x A B)
                   (and (in? x A) (not (in? x B))))))
```

Příklad (Konstruktory pro relace)

```
(define make-tuple cons)

(define cartesian-square
  (lambda (set)
    (apply append
           (map (lambda (x)
                  (map (lambda (y)
                         (make-tuple x y))
                       set))
                set))))

(define make-relation
  (lambda (prop? universe)
    (filter (lambda (x)
              (prop? (car x) (cdr x)))
            (cartesian-square universe))))
```

Příklad (Vytváření seznamů reprezentující binární relace)

```
(define u '(0 1 2 3 4 5))
(make-relation (lambda (x y) #f) u)
  ⟹  ()
(make-relation (lambda (x y) (= x y)) u)
  ⟹  ((0 . 0) (1 . 1) (2 . 2) (3 . 3) (4 . 4) (5 . 5))
(make-relation (lambda (x y) (= (+ x 1) y)) u)
  ⟹  ((0 . 1) (1 . 2) (2 . 3) (3 . 4) (4 . 5))
(make-relation (lambda (x y)
                 (= (modulo (+ x 1) (length u)) y)) u)
  ⟹  ((0 . 1) (1 . 2) (2 . 3) (3 . 4) (4 . 5) (5 . 0))
(make-relation (lambda (x y) (< x y)) u)
  ⟹  ((0 . 1) (0 . 2) (0 . 3) (0 . 4) (0 . 5)
       (1 . 2) (1 . 3) (1 . 4) (1 . 5) (2 . 3) (2 . 4)
       (2 . 5) (3 . 4) (3 . 5) (4 . 5))
```

Příklad (Reprezentace relací: další problémy)

Operace s relacemi:

- množinové operace (relace jsou množiny) – již máme
- `compose-relations` – kompozice (skládání) relací
- `invert-relation` – inverze relace

Vlastnosti binárních relací na množině:

- `reflexive?` – test reflexivity relace
- `symmetric?` – test symetrie relace
- `transitive?` – test tranzitivity relace

⋮

Skripta (sekce 6.5, strany 161–167)