KATEDRA INFORMATIKY, PŘÍRODOVĚDECKÁ FAKULTA
UNIVERZITA PALACKÉHO, OLOMOUC

PARADIGMATA PROGRAMOVÁNÍ 2
AKTUÁLNÍ POKRAČOVÁNÍ

Slajdy vytvořili Vilém Vychodil a Jan Konečný

Motivace

Během posledních dvou semestrů jsme ukázali, že s procedurami a makry je možné pracovat jako s daty. Dále jsme ukázali, že výpočet lze řídit daty (streamy). Nyní ukážeme, že s *výpočetním časem, historií a budoucností výpočtu* je možné pracovat jako s daty.

Budeme potřebovat tři fundamentální pojmy

1. *Kontext* – procedura jednoho argumentu reprezentující výpočet, který nastane okamžitě po vyhodnocení jistého výrazu
2. *Úniková procedura* – procedura, po jejíž aplikaci se ukončí zbylý výpočet a jako výsledek je okamžitě vrácena hodnota její aplikace
3. *Aktuální pokračování* neboli *kontinuace* – je úniková procedura vytvořená z kontextu aktuálního výrazu

Motivace

O co jde?

- aktuální pokračování – analogie "příkazu skoku"
- oproti skoku je však **MNOHEM** mocnější
- umožní nám vytvářet množství typů nelokálních skoků
- umožní (do jisté míry) zhmotnit budoucnost výpočtu a manipulovat s ní jako s daty (například ji vyvolat v okamžiku, kdy už „budoucí výpočet" proběhl – v jistém smyslu se tedy vrátíme do minulosti).
- časem vytvoříme
  - *korutiny* – podprogramy, které se budou vzájemně přepínat
  - *paralelní systém* – souběžný běh několika výpočtů
  - *nedeterministické operátory* – povede na programy, které budou schopny samy hledat „řešení problému" pouze na základě jeho popisu
  - a mnohem víc...
- aktuální pokračování – doména jazyka Scheme
  (ostatní PJ mají jen „trapné náhražky",
  třeba standard POSIX definuje `longjmp`... slabý odvar)

POJEM 1: Kontext

Neformálně: *„Kontext je zhmotnění zbytku výpočtu, který by byl zahájen okamžitě po vyhodnocení nějakého podvýrazu“*

Příklady:

kontext podvýrazu `(/ 25 x)` ve výrazu `(+ 2 (* 3 (/ 25 x)))` je výpočet, který proběhne po vyhodnocení `(/ 25 x)` v `(+ 2 (* 3 (/ 25 x)))`, to jest: hodnota vzniklá vyhodnocením `(/ 25 x)` bude násobena číslem `3` a tento výsledek bude přičten k `2`.

kontext podvýrazu `*` ve výrazu `(+ 2 (* 3 (/ 25 x)))` je výpočet, který nejprve otestuje, zda-li je výsledná hodnota (vzniklá vyhodnocením `*`) procedura, pokud ne, končí se chybou; pokud ano, je tato procedura aplikována na `3` a výsledek vyhodnocení `(/ 25 x)`; k výsledku aplikace je poté ještě přičteno `2`.

Ještě něco o kontextu

Pojem: *program s dírou* (hole „□“)
díra představuje (nespecifikovaný) výraz, o jehož kontext se zajímáme

```
(+ 2 (* 3 (/ 25 5)))
(+ 2 (* 3 □))
(+ 2 (* □ (/ 25 5)))
(+ 2 (□ 3 (/ 25 5)))
```

Formálně: Kontext je **procedura jednoho argumentu** reprezentující výpočet, který by nastal po dosazení skutečné hodnoty místo díry

Pro předchozí příklady si lze představit jako procedury vzniklé vyhodnocením následujících λ-výrazů

```
(lambda (□) (+ 2 (* 3 □)))
(lambda (□) (+ 2 (* □ (/ 25 5))))
(lambda (□) (+ 2 (□ 3 (/ 25 5))))
```

POJEM 2: Úniková procedura

Procedura se nazývá *úniková procedura*, pokud je-li aktivována, tak způsobí okamžité přerušení vykonávání zbytku výpočtu a výsledkem vyhodnocení (celého vstupního výrazu, ve kterém byla použita) je právě výsledek její aplikace

Budeme předpokládat, že máme k dispozici proceduru `escaper`, která pro danou proceduru vrátí tutéž proceduru, která ale bude úniková (posléze ukážeme, že `escaper` lze ve Scheme naprogramovat)

`*` $\longmapsto$ procedura násobení
`(escaper *)` $\longmapsto$ úniková procedura násobeni

Použití na top-level je stejné

```
(* 10 20)                 ⟼ 200
((escaper *) 10 20)       ⟼ 200, avšak:

(+ 2 (* 10 20))           ⟼ 202
(+ 2 ((escaper *) 10 20)) ⟼ 200
```

Další příklad únikové funkce

```
(define p (escaper
           (lambda (x)
             (if (even? x)
                 x
                 (- x)))))
```

příklad použití:

```
(p 10)  ⟹  10
(p 11)  ⟹  -11
```

```
(define p (escaper
           (lambda (x)
             (if (even? x)
                 x
                 (- x)))))

(define test
  (lambda (x)
    (if (= 1 (p x))
        tohle-nikdy-neprobehne
        tohle-taky-ne)))
```

Demonstrace toho, že `if` nikdy neproběhne

```
(test 10)                  ⟹  10
(test 11)                  ⟹  -11
(eval '(test 10))          ⟹  10
(eval '(test (+ 5 6)))     ⟹  -11
```

POJEM 3: Aktuální pokračování (kontinuace)

Jazyk Scheme dává k dispozici proceduru `call/cc`, použití:

`(call/cc` ◁*receiver*▷`)`,

kde ◁*receiver*▷ (příjemce) musí být *procedura jednoho argumentu.*
(`call/cc` je zkratka pro `call-with-current-continuation`)

Při volání `(call/cc` ◁*receiver*▷`)` **se provede:**

1. Vytvoří se kontext `(call/cc` ◁*receiver*▷`)` v aktuálně vyhodnocovaném výrazu
2. ◁*receiver*▷ je zavolán s argumentem (který nazýváme **aktuální pokračování** neboli **kontinuace**) jímž je procedura vzniklá vyhodnocením `(escaper` ◁*kontext*▷`)`

Takže při použití

```
··· (call/cc
      (lambda (f)
        ···
        (f ···)
        ···)) ···
```

bude uvnitř na `f` navázána *úniková procedura.*

Tuto únikovou proceduru můžeme aktivovat s jedním argumentem.

Pokud se tak stane, je okamžitě přerušeno vyhodnocování dalšího kódu v receiveru a bude se pokračovat vyhodnocováním kontextu `(call/cc ···)` s hodnotou se kterou bylo aplikováno `f`.

```
(* 2 (+ 3 (/ 25 5)))  ⟹  16
```

Takže například (f není použito)

```
(* 2 (call/cc
      (lambda (f)
        (+ 3 (/ 25 5)))))  ⟹  16
```

předchozí si lze představit:

```
(* 2 ((lambda (f)
        (+ 3 (/ 25 5)))
      (escaper (lambda (□) (* 2 □)))))
```

Další příklad:

```
(* 2 (call/cc
      (lambda (f)
        (+ 3 (f 5)))))  ⟹  10
```

si lze představit:

```
(* 2 ((lambda (f)
        (+ 3 (f 5)))
      (escaper (lambda (□) (* 2 □)))))
```

u předchozího: `(+ 3 ...` se neuplatní,
dojde k přerušení výpočtu a vyvolá se
aktuální pokračování (násobení dvojkou)

```
(+ 1 (call/cc (lambda (f) 2)))  ⟼  3

(+ 1 (call/cc (lambda (f) (f 2))))  ⟼  3

(+ 1 (call/cc (lambda (f)
                (if (even? (f 2)) 100 200))))  ⟼  3

(+ 1 (call/cc
      (lambda (f)
        (* 2 (call/cc
              (lambda (g)
                (f (g 20)))))))))  ⟼  41

(+ 1 (call/cc
      (lambda (f)
        (* 2 (call/cc
              (lambda (g)
                (g (f 20)))))))))  ⟼  21
```

Aplikace

- okamžité opuštění rekurze (výskok z rekurze)

Modelový příklad

```
;; procedura realizující součin čísel v seznamu
(define list-product
  (lambda (l)
    (if (null? l)
        1
        (* (car l) (list-product (cdr l))))))
```

```
(list-product '(1 2 3 4 5))  ⟼  120
```

- chceme zefektivnit výpočet
- násobení čehokoliv nulou je nula

přidání limitní podmínky pro nulu

má nevýhodu, fáze *odvíjení* bude pořád probíhat

```
(define list-product
  (lambda (l)
    (cond ((null? l) 1)
          ((= (car l) 0) 0)
          (else (* (car l) (list-product (cdr l)))))))

;; rekurzivní verze procedury používající call/cc
(define list-product
  (lambda (l)
    (call/cc
     (lambda (exit)
       (let proc ((l l))
         (cond ((null? l) 1)
               ((= (car l) 0) (exit 0))
               (else (* (car l) (proc (cdr l)))))))))))
```

```
;; iterativní verze procedury
(define list-product
  (lambda (l)
    (let iter ((l l)
               (accum 1))
      (if (null? l)
          accum
          (iter (cdr l) (* accum (car l)))))))

;; iterativní verze s okamžitým opuštěním (bez call/cc)
(define list-product
  (lambda (l)
    (let iter ((l l)
               (accum 1))
      (cond ((null? l) accum)
            ((= (car l) 0) 0)
            (else (iter (cdr l) (* accum (car l))))))))
```

Poznámka

- V předchozím případě jsme zefektivnění výpočtu pomocí okamžitého opuštění uměli udělat bez `call/cc`, pouze jsme *rekurzivní proceduru* nahradili její *iterativní variantou* (iteraci lze kdykoliv okamžitě přerušit, protože se nebuduje žádný „nedokončený výpočet“).
- Otázka: „Není `call/cc` jen zbytečný luxus?“
  Odpověď: *není*, přepis rekurze pomocí iterace je někdy hodně těžký (například iterativní verze hloubkově rekurzivních procedur)
- viz následující odstrašující příklad

```
;; testuj vztah dvou seznamů do hloubky
(define atom-prop?
  (lambda (aprop? l1 l2)
    (cond ((and (null? l1) (null? l2)) #t)
          ((and (pair? l1) (pair? l2))
           (and (atom-prop? aprop? (car l1) (car l2))
                (atom-prop? aprop? (cdr l1) (cdr l2))))
          ((and (not (pair? l1)) (not (pair? l2)))
           (aprop? l1 l2))
          (else #f))))
```

Příklad použití:

```
(atom-prop? <= '(1 (2 (3) 4)) '(2 (3 (4) 5)))  ⟹  #t
(atom-prop? <= '(1 (2 (3) 4)) '(2 (3 (1) 5)))  ⟹  #f
```

```
;; nebo bez pomocí cond (jen s použitím and a or)
(define atom-prop?
  (lambda (aprop? l1 l2)
    (or (and (null? l1)
             (null? l2))
        (and (pair? l1)
             (pair? l2)
             (atom-prop? aprop? (car l1) (car l2))
             (atom-prop? aprop? (cdr l1) (cdr l2)))
        (and (not (pair? l1))
             (not (pair? l2))
             (aprop? l1 l2)))))
```

```
;; efektivnější verze s okamžitým opuštěním výpočtu
(define atom-prop?
  (lambda (aprop? l1 l2)
    (call/cc
     (lambda (exit)
       (let test ((l1 l1)
                  (l2 l2))
         (or (and (null? l1)
                  (null? l2))
             (and (pair? l1)
                  (pair? l2)
                  (test (car l1) (car l2))
                  (test (cdr l1) (cdr l2)))
             (and (not (pair? l1))
                  (not (pair? l2))
                  (aprop? l1 l2))
             (exit #f)))))))
```

- předchozí procedur lze přepsat i iterativně, ale je to složité
- navíc se poněkud vytrácí efektivita, protože rekurzivní volání procedury nahrazujeme komplikovanou manipulací se zásobníky

```
(define atom-prop?
  (lambda (aprop? l1 l2)
    (let test ((s1 '())  ; pomocný zásobník pro l1
               (s2 '())  ; pomocný zásobník pro l2
               (l1 l1)
               (l2 l2))
      (cond ((and (null? s1) (null? s2)
                  (null? l1) (null? l1)) #t)
            ((and (null? s1) (null? s2)
                  (not (null? l1)) (not (null? l2)))
             (test (cons (car l1) s1)
                   (cons (car l2) s2)
                   (cdr l1)
                   (cdr l2)))
            ((and (null? s1) (null? s2)) #f)  ...
```

```
  ⋮
((or (null? s1) (null? s2)) #f)
((and (null? (car s1)) (null? (car s2)))
 (test (cdr s1) (cdr s2) l1 l2))
((and (pair? (car s1)) (pair? (car s2)))
 (test
  (cons (caar s1) (cons (cdar s1) (cdr s1)))
  (cons (caar s2) (cons (cdar s2) (cdr s2)))
  l1
  l2))
((or (pair? (car s1)) (pair? (car s2))) #f)
((aprop? (car s1) (car s2))
 (test (cdr s1) (cdr s2) l1 l2))
(else #f)))))
```

```
(atom-prop? <= '((1 2) (3)) '((2 3) (4)))  ⟹  #t
[] [] ((1 2) (3)) ((2 3) (4))
[(1 2)] [(2 3)] ((3)) ((4))
[1 (2)] [2 (3)] ((3)) ((4))
[(2)] [(3)] ((3)) ((4))
[2 ()] [3 ()] ((3)) ((4))
[()] [()] ((3)) ((4))
[] [] ((3)) ((4))
[(3)] [(4)] () ()
[3 ()] [4 ()] () ()
[()] [()] () ()
[] [] () ()
(atom-prop? <= '((1 2) (3)) '((1 1) (4)))  ⟹  #f
[] [] ((1 2) (3)) ((1 1) (4))
[(1 2)] [(1 1)] ((3)) ((4))
[1 (2)] [1 (1)] ((3)) ((4))
[(2)] [(1)] ((3)) ((4))
[2 ()] [1 ()] ((3)) ((4))
```

Úschova únikových funkcí a jejich pozdější použití

Kontinuace jsou (únikové) procedury

Kontinuace = elementy prvního řádu

```
(call/cc (lambda (f) f))                    ⟼ #<continuation>
(procedure? (call/cc (lambda (f) f)))       ⟼ #t
```

Receiver v následujícím výrazu nejprve naváže kontinuaci na globální symbol `blah`

```
(define blah #f)
(* 2 (call/cc
       (lambda (f)
         (set! blah f)
         30)))   ⟼   60
```

Kontinuaci v `blah` je možné dál používat

```
(blah 1)   ⟼   2
(blah 2)   ⟼   4
(blah 3)   ⟼   6
```

Úschova únikových funkcí a jejich pozdější použití

```
(begin
  (display "JEDNOU")
  (newline)
  (* 2 (call/cc
        (lambda (f)
          (set! blah f)
          30))))  ⟼  60 a vytiskne se JEDNOU
(blah 10)  ⟼  20 (na obrazovku se nic netiskne)

(* 2 (let ((result (call/cc
                    (lambda (f)
                      (set! blah f)
                      30))))
       (display "POKAZDE")
       (newline)
       result))  ⟼  60
(blah 10)  ⟼ 20 a vytiskne se POKAZDE
```

Úschova únikových funkcí a jejich pozdější použití

Zajímavý efekt

```
(* 2 (call/cc
      (lambda (f)
        (set! blah f)
        zde-udelame-zamerne-chybu)))
```
$\Longmapsto$ Error

Chyba nastala až po navázání `blah`, takže:

`(blah 10)` $\Longmapsto$ 20 (je v pořádku)

Příklad: Implementace `escaper` pomocí `call/cc`

Globální symbol na kterém budeme mít
navázanou kontinuaci, viz další výraz

```
(define *top-level-escaper* #f)
```

Na `*top-level-escaper*` naváže
další pokračování cyklu REPL:

```
(call/cc
 (lambda (break)
   (set! *top-level-escaper* break)))
```

Následující je implementace `escaper` pomocí `call/cc`

```
(define escaper
  (lambda (f)
    (lambda args
      (*top-level-escaper* (apply f args)))))
```

Implementace chybového hlášení

```
(define error
  (lambda (proc message . values)
    (display "Error ") (display (symbol->string proc))
    (display ": ") (display message)
    (if (not (null? values))
        (begin
          (display " ")
          (let ((first #t))
            (for-each
             (lambda (x)
               (if first (set! first #f) (display ", "))
               (display x))
             values))
          (display "")))
    (newline)
    (*top-level-escaper*)))
```

Příklad použití

```
(define list-product
  (lambda (l)
    (call/cc
     (lambda (exit)
       (let proc ((l l))
         (cond ((null? l) 1)
               ((not (pair? l))
                (error 'list-product
                       "Given argument is not a list"
                       l))
               ((not (number? (car l)))
                (error 'list-product
                       "List member is not a number"
                       (car l)))
               ((= (car l) 0) (exit 0))
               (else (* (car l) (proc (cdr l)))))))))))
```

Příklad: Systém výjimek (procedura `throw` a makro `catch`)

```
(define throw
  (lambda (excptn-name expr)
    (error 'throw "Top level throw triggered with"
                  excptn-name)))

(define-macro catch
  (lambda (label . prgn)
    (let ((exit (gensym)))
      `(call/cc
        (lambda (,exit)
          (let ((throw (lambda (excptn-name expr)
                         (if (eq? ,label excptn-name)
                             (,exit expr)
                             (throw excptn-name expr)))))
            (begin ,@prgn)))))))
```

Příklad použití

```
(catch 'chyba
  (+ 1 2)
  (* 3 4))  ⟼  12
```

Příklad použití

```
(catch 'chyba
  (+ 1 2)
  (throw 'chyba 20)
  (* 3 4))  ⟼  20
```

Příklad použití

```
(catch 'chyba
  (+ 1 2)
  (throw 'neexistujici-vyjimka 20)
  (* 3 4))  ⟼ Error
```

Další příklad

```
(define f
  (lambda (n)
    (let ((x 10))
      (catch 'bar
             (catch 'foo
                    (if (> n 0)
                        (throw 'foo (set! x 40)))
                    (if (< n 0)
                        (throw 'bar (+ 1 2)))
                    (set! x 100))
             (* 2 x)))))
```

```
(f -1)  ⟼  3
(f 0)   ⟼  200
(f 1)   ⟼  80
```

Další příklad ošetřování výjimečných situací: aserce

```
(define-macro assert
  (lambda (proc-name assertions . body)
    `(cond ,@(map (lambda (ass)
                    `((not ,(car ass))
                      (begin
                        (display "ASSERT ")
                        (display ,proc-name)
                        (display ": ")
                        ,@(map (lambda (x)
                                 `(display ,x))
                               (cdr ass))
                        (newline)
                        (*top-level-escaper*))))
                  assertions)
           (else (begin ,@body)))))
```

Příklad: faktoriál s asercemi

```
(define fak
  (lambda (n)

    (assert "faktorial"
      (((number? n) "arg. must be a number, given: " n)
       ((exact? n) n " must be an exact number")
       ((>= n 0) n " must be a non-negative integer"))

      (if (= n 0)
          1
          (* n (fak (- n 1)))))))
```

Po odladění programu je možné `assert` nahradit:

```
(define-macro assert
  (lambda (proc-name assertions . body)
    `(begin ,@body)))
```

Příklad: Jednoduchý debugger

Globální symboly na kterých jsou navázány break-point a zastavení

```
(define *next* #f)
(define *stop* #f)
```

Nastavení stopky

```
(call/cc
 (lambda (f)
   (set! *stop* f)))
```

Přeruš výpočet

```
(define stop-execution
  (lambda ()
    (*stop*)))
```

Příklad: Jednoduchý debugger

Zapiš `*next*` aktuální pokračování v daném bodě výpočtu, ve kterém bude zavolána procedura `break-point`

```
(define break-point
  (lambda ()
    (call/cc
     (lambda (f)
       (set! *next* f)
       (stop-execution)
       hic-sunt-leones))))
```

Pokračuj ve výpočtu (až k dalšímu breakpointu)

```
(define run
  (lambda ()
    (*next*)))
```

Pro ukázku viz soubor `debugger.scm`

Příklad: Iterátory a jejich aplikace.

- iterátor – pro danou datovou strukturu postupně vrací její prvky

Pomocné definice:

Identifikátor ukončení iterace

```
(define *end-of-iteration* (lambda () #f))
```

Implementace chybového hlášení
Predikát indikující konec iterace

```
(define finished?
  (lambda (elem)
    (eq? elem *end-of-iteration*)))
```

```
(define generate-iterator
  (lambda (l)
    (letrec ((return #f)
             (start
              (lambda ()
                (let loop ((l l))
                  (if (null? l)
                      (return *end-of-iteration*)
                      (begin
                        (call/cc
                         (lambda (new-start)
                           (set! start new-start)
                           (return (car l))))
                        (loop (cdr l)))))))))
      (lambda () (call/cc
                  (lambda (f)
                    (set! return f)
                    (start)))))))
```

Příklad použití:

```
(define p (generate-iterator '(a b c d e)))
```

```
(p)  ⟹ a
(p)  ⟹ b
(p)  ⟹ c
(p)  ⟹ d
(p)  ⟹ e
(p)  ⟹ #<procedura> (indikátor konce)
(eq? (p) *end-of-iteration*)  ⟹  #t
```

Příklad: Hloubkový iterátor

```
(define generate-depth-iterator
  (lambda (l)
    (letrec ((return #f)
             (start
              (lambda ()
                (let loop ((l l))
                  (cond ((null? l) 'nejaka-hodnota)
                        ((pair? l)
                         (begin
                           (loop (car l))
                           (loop (cdr l))))
                        (else
                         (call/cc
                          (lambda (new-start)
                            (set! start new-start)
                            (return l))))))))
             (return '()))))) ...
```

```
  ⋮
(lambda ()
  (call/cc
   (lambda (f)
     (set! return f)
     (start))))))
```

Poznámka

- už není potřeba `*end-of-iteration*`
- prohledávání je ukončeno `()`

Příklad použití

```
(define p (generate-depth-iterator '(a (b (c (d)) e))))
(define q (generate-depth-iterator '(((a b) ((c d))) e)))
(p)  ⟹  a
(q)  ⟹  a
(p)  ⟹  b
(q)  ⟹  b
(p)  ⟹  c
(q)  ⟹  c
(p)  ⟹  d
(q)  ⟹  d
(p)  ⟹  e
(q)  ⟹  e
(p)  ⟹  ()
(q)  ⟹  ()
```

Zvýšení výpočetní efektivity pomocí iterátorů

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

Nová, efektivní verze `equal-fringe?`

```
;; procedura používá iterátory
(define equal-fringe?
  (lambda (s1 s2)
    (let ((iter1 (generate-depth-iterator s1))
          (iter2 (generate-depth-iterator s2)))
      (let test ((v1 (iter1))
                 (v2 (iter2)))
        (or (and (null? v1) (null? v2))
            (and (equal? v1 v2)
                 (test (iter1) (iter2)))))))))
```