KATEDRA INFORMATIKY, PŘÍRODOVĚDECKÁ FAKULTA
UNIVERZITA PALACKÉHO, OLOMOUC

PARADIGMATA PROGRAMOVÁNÍ 2
PŘÍSLIBY A LÍNÉ VYHODNOCOVÁNÍ

Slajdy vytvořili Vilém Vychodil a Jan Konečný

Přísliby a líné vyhodnocování

Základní myšlenka

- místo vyhodnocení daného výrazu pracujeme s **příslibem** jeho **budoucího vyhodnocení**
- **příslib** = nový typ elementu (element prvního řádu)

Co je potřeba k tomu, aby to fungovalo:

1. k dispozici je spec. forma (nejčastěji zvaná `delay`), která pro *daný výraz* vrací *příslib jeho vyhodnocení*
2. k dispozici je procedura (nejčastěji zvaná `force`), která pro *daný příslib* aktivuje výpočet a vrátí hodnotu vzniklou vyhodnocením přislíbeného výrazu

Líné vyhodnocování:

- vyhodnocování založené na příslibech
- někdy se nazývá «call by need»

Příklad zamýšleného použití

```
(delay (+ 1 2))  ⟹  #<promise>
(define p (delay (map - '(1 2 3 4))))
p                ⟹  #<promise>
(promise? p)     ⟹  #t
(force p)        ⟹  (-1 -2 -3 -4)
```

Poznámky:

- `delay` nemůže být z principu procedura, protože chceme, aby se přislíbený výraz vyhodnotil až při aktivaci pomocí `force`
- při líném vyhodnocování dochází k *propagaci chyb* (chyba se projeví „na jiném místě“ než „kde vznikla“)

```
(define p (delay blah))   proběhne bez problémů
p                         ⟹  #<promise>
⋮
(force p)                 ⟹  CHYBA: blah nemá vazbu
```

- pomocí příslibů je možné „odložit časově složitý výpočet na později“ a aktivovat jej, až je skutečně potřeba jej provést

Modelový časově náročný výpočet:

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

Příklad použití

```
(define p (delay (fib 30)))
```

proběhne okamžitě

⋮

```
(force p)
```

aktivace výpočtu (prodleva)

Při použití příslibů vyvstávají otázky spojené s **vedlejším efektem**.

Příslib výrazu, který má vedlejší efekt:

```
(define p (let ((i 0))
            (delay (begin
                     (set! i (+ i 1))
                     i))))
```

Dvojí aktivace výpočtu:

```
(force p)  ⟹  1
(force p)  ⟹  ???
```

Možnosti:

1. druhá aktivace `(force p)` vrací 1
2. druhá aktivace `(force p)` vrací 2, třetí vrací 3, . . .

Ukážeme, jak implementovat líné vyhodnocování umožňující obě varianty.

Implementace příslibů a líného vyhodnocování

- přísliby lze plně implementovat pomocí *procedur vyšších řádů*, *maker* a *vedlejších efektů*

Základní myšlenka:

- při vytváření procedur (vyhodnocováním λ-výrazů) nedochází k vyhodnocování těla nově vznikajících procedur
- k vyhodnocování těla procedur dochází až při jejich aplikaci
- nabízí se tedy: *vytvořit přísliby pomocí procedur*

Vysvětlující příklad

```
(lambda () (+ 1 2))                  #<procedure>
(define p (lambda () (+ 1 2)))       náš příslib
(p)                                  aktivace
```

Implementace příslibů a líného vyhodnocování

- jednodušší verze
- při každé aktivaci příslibu je přislíbený výraz vždy vyhodnocen
- vytvoříme makro `freeze` („zmraz“) a proceduru `thaw` („roztaj“)

```
;; speciální forma freeze
(define-macro freeze
  (lambda exprs
    `(lambda ()
       (begin ,@exprs))))

;; procedura thaw
(define thaw
  (lambda (promise)
    (promise)))
```

Příklad vytvoření příslibu

```
(define p
  (let ((x 10))
    (freeze (display "Hodnota: ")
            (display x)
            (newline)
            (set! x (+ x 1))
            (list x (* x x)))))
```

Příklad aktivace příslibu

```
(thaw p)  ⟹  (11 121)
(thaw p)  ⟹  (12 144)
(thaw p)  ⟹  (13 169) ···
```

Příslib jako součást složitějšího výrazu

```
(reverse (thaw p))  ⟹  (225 14)
```

Implementace příslibů a líného vyhodnocování

- složitější verze
- při první aktivaci příslibu je výsledek vyhodnocení přislíbeného výraz **zapamatován** (uvnitř příslibu) a při každé další aktivaci příslibu je *vrácena zapamatovaná hodnota*
- případné *vedlejší efekty* se projeví *jen při první aktivaci*
- vytvoříme makro `delay` a proceduru `force`

Nejprve příklad použití:

```
(define p
  (let ((x 10))
    (delay (set! x (+ x 1))
           (list x (* x x)))))

(force p)  ⟹  (11 121)
(force p)  ⟹  (11 121) ···
```

Implementace příslibů a líného vyhodnocování

Speciální forma `delay`

```
(define-macro delay
  (lambda exprs
    `(let ((result (lambda ()
                     (begin ,@exprs)))
           (evaluated? #f))
       (lambda ()
         (begin
           (if (not evaluated?)
               (begin
                 (set! evaluated? #t)
                 (set! result (result))))
           result)))))
```

```
;; procedura force (totéž co thaw)
(define force thaw)
```

Proudy (angl. Streams)

- proudy jsou nejčastěji používanou *aplikací líného vyhodnocování*
- neformálně: proudy jsou *líně vyhodnocované seznamy*
- konstruktor `cons-stream` a selektory `stream-car` a `stream-cdr`

```
;; Konstruktor proudu cons-stream je makro
(define-macro cons-stream
  (lambda (a b)
    `(cons ,a (delay ,b))))

;; selektor stream-car (vrať první prvek proudu)
(define stream-car car)

;; selektor stream-cdr (vrať proud bez prvního prvku)
(define stream-cdr
  (lambda (stream)
    (force (cdr stream))))
```

Definice proudů

- prázdný seznam je proud;
- každý tečkový pár $(e\ .\ p)$, kde
  e je libovolný element a p je příslib proudu, je proud.

```
;; je stream prázdný?
(define stream-null? null?)

;; predikát stream? (podle definice)
(define stream?
  (lambda (elem)
    (or (null? elem)
        (and (pair? elem)
             (and (promise? (cdr elem))
                  (stream? (force (cdr elem))))))))
```

- předchozí predikát `stream?` má nevýhodu: používá `force` (`!`)

```
;; slabší verze predikátu stream?
;; každý pár, jehož 2. prvek je příslib nebo () je stream
(define stream?
  (lambda (elem)
    (or (null? elem)
        (and (pair? elem)
             (or (promise? (cdr elem))
                 (null? (cdr elem)))))))
```

Pomocné procedury:

```
;; zobraz stream, nanejvýš však n prvních prvků
(define display-stream (lambda (stream . n) ···

;; odvozené selektory
(define stream-caar (lambda (x) ···
  ⋮
(define stream-cddddr (lambda (x) ···
```

Procedury pro práci s proudy

```
;; délka proudu
(define stream-length
  (lambda (stream)
    (if (stream-null? stream)
        0
        (+ 1 (stream-length (stream-cdr stream))))))

;; mapování přes proudy (mapování přes jeden stream)
(define stream-map2
  (lambda (f stream)
    (if (stream-null? stream)
        '()
        (cons-stream
         (f (stream-car stream))
         (stream-map f (stream-cdr stream))))))
```

Procedury pro práci s proudy

```
;; mapování přes proudy (obecná verze)
(define stream-map
  (lambda (f . streams)
    (if (stream-null? (car streams))
        '()
        (cons-stream
         (apply f (map stream-car streams))
         (apply stream-map f
                (map stream-cdr streams))))))

;; konvertuj seznam na stream
(define list->stream
  (lambda (list)
    (foldr (lambda (x y)
             (cons-stream x y))
           '() list)))
```

Procedury pro práci s proudy

```
;; vytvoř konečný stream daných prvků
(define stream
  (lambda args
    (list->stream args)))
```

Příklady použití předchozích procedur:

```
(define s (stream 1 2 3 4))
s                          ⟼  (1 . #<promise>)
(display-stream s)         ⟼  #<stream (1 2 3 4)>
(display-stream s 2)       ⟼  #<stream (1 2)>
(stream-length s)          ⟼  4
(display-stream (stream-map - s) 2)
  ⟼ #<stream (-1 -2)>
(display-stream (stream-map + s s s))
  ⟼  #<stream (3 6 9 12)>
```

Všimněte si

- při práci s proudy mají jednotlivé procedury „jinou odezvu“
- výpočet je řízen daty, dochází k propagaci chyb

Výsledek je vrácen okamžitě

```
(define fs (stream-map fib (stream 1 ··· 30 31 ··· 50)))
fs  ⟼  (1 . #<promise>)
```

přístup ke dalším prvkům se bude postupně zpomalovat

ukázka propagace chyb v proudech:

```
(define s (stream 1 2 3 4 5 'blah 6 7))
(define r (stream-map - s))
r                        ⟼  (-1 . #<promise>)
(display-stream r 4)     ⟼  (-1 -2 -3 -4)
(display-stream r 6)     ⟼  (-1 -2 -3 -4 -5 CHYBA
```

Úskalí

```
;; zdánlivě funkční verze 'foldr' pro proudy
(define stream-foldr
  (lambda (f nil . streams)
    (if (stream-null? (car streams))
        nil
        (apply f
         `(,@(map stream-car streams)
           ,(apply stream-foldr f nil
                   (map stream-cdr streams)))))))

;; následující se nechová přirozeně
(stream-foldr (lambda (x y) (cons-stream (- x) y))
              '() (stream 1 2 3 'blah 4))
```

$\longmapsto$ CHYBA: nelze aplikovat `-` na symbol `blah`

Čekali bychom, že chyba se projeví až při pokusu přistoupit ke 4. prvku výsledného proudu

Nová verze `stream-foldr`

- proceduře „`f`“ bude předáván místo druhého argumentu jeho příslib
- procedura sama rozhodne, jak bude s příslibem nakládat

```
;; procedura stream-folder
(define stream-foldr
  (lambda (f nil . streams)
    (if (stream-null? (car streams))
        nil
        (apply f
         `(,@(map stream-car streams)
             ,(delay
                 (apply stream-foldr f nil
                        (map stream-cdr streams))))))))

(stream-foldr (lambda (x y) (cons-stream (- x) (force y)))
              '() (stream 1 2 3 'blah 4))
  ⟹  (-1 . #<promise>)
```

Další (užitečné) odvozené procedury

```
;; konverze proudu na seznam
(define stream->list
  (lambda (stream)
    (stream-foldr (lambda (x y)
                    (cons x (force y)))
                  '()
                  stream)))

;; filtrace prvku proudu podle vlastnosti
(define stream-filter
  (lambda (prop? stream)
    (stream-foldr (lambda (x y)
                    (if (prop? x)
                        (cons-stream x (force y))
                        (force y)))
                  '()
                  stream)))
```

Nekonečné proudy a jejich implicitní definice

Příklad (proud jedniček)

```
;; rekurzivní procedura bez limitní podmínky
(define ones-proc
  (lambda ()
    (cons-stream 1 (ones-proc))))

;; nekonečný proud vytvořený voláním ones-proc
(define ones (ones-proc))

;; předchozí s použitím pojmenovaného let
(define ones (let proc ()
               (cons-stream 1 (proc))))

;; implicitní definice proudu
(define ones (cons-stream 1 ones))
```

Nekonečné proudy a jejich implicitní definice

Příklad (proud přirozených čísel)

```
;; rekurzivní procedura bez limitní podmínky
(define naturals-proc
  (lambda (i)
    (cons-stream i (naturals-proc (+ i 1)))))

;; nekonečný proud vytvořený voláním naturals-proc
(define naturals (naturals-proc 1))

;; předchozí s použitím pojmenovaného let
(define naturals (let iter ((i 1))
                   (cons-stream i (iter (+ i 1)))))

;; implicitní definice proudu
(define naturals (cons-stream 1
                   (stream-map + ones naturals)))
```

Nekonečné proudy

Nekonečný proud

- neformálně: „potenciálně nekonečná lineární datová struktura“
- *potenciálně nekonečná* znamená:
  - opakovaným použitím `stream-cdr` se nedostaneme na jejich konec
  - v každém okamžiku průchodu nekonečným proudem máme vždy k dispozici *aktuální prvek* a *příslib pokračování proudu*
- lze se na něj dívat jako na nekonečnou posloupnost elementů $(e_i)_{i=0}^{\infty}$, to jest $e_0, e_1, e_2, \ldots, e_{n-1}, e_n, e_{n+1}, \ldots$
- v praxi se konstruuje rekurzivní procedurou *bez limitní podmínky*

```
;; proud hodnot (2^i)_{i=0}^∞
(define pow2
  (let next ((last 1))
    (cons-stream last
                 (next (* 2 last)))))
```

Nekonečné proudy

Formálně lze zavést jako limity prefixových generátorů

Seznam r je **prefix** seznamu t, pokud lze t vyjádřit jako spojení r s nějakým seznamem l (v tomto pořadí).

Množinu seznamů $\mathcal{S}$ nazveme **prefixový generátor**, pokud

1. pro každé $n \in \mathbb{N}$, systém $\mathcal{S}$ obsahuje seznam délky n;
2. pro každé dva $s, t \in \mathcal{S}$ platí: buď s je prefix t, nebo t je prefix s.

Nekonečný proud (příslušný prefixovému generátoru $\mathcal{S}$) je element reprezentující posloupnost $(e_i)_{i=0}^{\infty}$, kde e_i je element nacházející se na i-té pozici libovolného seznamu $s \in \mathcal{S}$ majícího alespoň $i + 1$ prvků.