

Paradigmata programování 1

Čistě funkcionální interpret Scheme

Vilém Vychodil

Katedra informatiky, PřF, UP Olomouc

Přednáška 12

Přednáška 12: Přehled

1 Scheme ve Scheme – základní úvahy

- interpret a metainterpret, ...
- elementy a metaelementy, ...
- systém manifestovaných typů
- reprezentace základních datových typů

2 Vyhodnocovací proces

- implementace `read`, `eval`, `print`
- reprezentace uživatelských a primitivních procedur
- reprezentace prostředí
- aplikace procedur a speciálních forem

3 Hierarchie prostředí

- tři úrovně počátečních prostředí
- možnosti a meze čistě funkcionálního Scheme

Čistě funkcionální interpret Scheme (ve Scheme)

Základní rys:

- během výpočtu *nedochází k žádným vedlejším efektům*

Co interpret umí:

- procedury: primitivní, uživatelské, procedury vyšších řádů
- elementy prvního řádu: čísla, symboly, seznamy, procedury, prostředí

Co interpret neumí:

- neumí (re)definovat vazbu symbolu (nemá *define*)
- rekurzivní procedury je potřeba zavádět pomocí *Y*-kombinátoru (to je nepohodlné, ale rekurzi lze používat bez omezení)
- nemá ani řadu dalších konstruktů (další semestry)

Systém manifestovaných typů

;; vytvoř element jazyka s manifestovaným typem

```
(define curry-make-elem  
  (lambda (type-tag)  
    (lambda (data)  
      (cons type-tag data))))
```

;; vrať visačku s typem, vrať data

```
(define get-type-tag car)  
(define get-data cdr)
```

;; vytvoř test pro daný datový typ

```
(define curry-scm-type  
  (lambda (type)  
    (lambda (elem)  
      (equal? type (get-type-tag elem)))))
```

Reprezentace čísel a symbolů

;; čísla

```
(define make-number (curry-make-elem 'number))  
(define scm-number? (curry-scm-type 'number))
```

;; symboly

```
(define make-symbol (curry-make-elem 'symbol))  
(define scm-symbol? (curry-scm-type 'symbol))
```

Příklad (vytvoření čísel a symbolů, test příslušného datového typu)

(make-number 10)	\Rightarrow	(number . 10)
(make-number -2.3)	\Rightarrow	(number . -2.3)
(make-symbol 'foo)	\Rightarrow	(symbol . foo)
(scm-number? (make-symbol 'foo))	\Rightarrow	#f
(scm-symbol? (make-symbol 'foo))	\Rightarrow	#t

Reprezentace speciálních elementů jazyka

;; pravdivostní hodnoty

```
(define scm-false ((curry-make-elem 'boolean) #f))
```

```
(define scm-true ((curry-make-elem 'boolean) #t))
```

```
(define scm-boolean? (curry-scm-type 'boolean))
```

;; prázdný seznam (je pouze jeden)

```
(define the-empty-list
```

```
  ((curry-make-elem 'empty-list) '()))
```

```
(define scm-null?
```

```
  (lambda (elem) (equal? elem the-empty-list)))
```

;; element zastupující nedefinovanou hodnotu (je pouze jeden)

```
(define the-undefined-value ((curry-make-elem 'undefined) '()))
```

```
(define scm-undefined?
```

```
  (lambda (elem) (equal? elem the-undefined-value)))
```

Reprezentace tečkových párů

;; konstruktor párů

```
(define make-pair  
  (let ((make-physical-pair (curry-make-elem 'pair)))  
    (lambda (head tail)  
      (make-physical-pair (cons head tail)))))
```

;; test pro datový typ páru

```
(define scm-pair? (curry-scm-type 'pair))
```

;; selektor `car` (selektor `cdr` se zavede analogicky)

```
(define pair-car  
  (lambda (pair)  
    (if (scm-pair? pair)  
        (car (get-data pair))  
        (error "; Car: argument must be a pair"))))
```


Převod metaelementů na odpovídající elementy

;; převed' vstupní symbolický výraz do interní reprezentace

```
(define expr->intern
  (lambda (expr)
    (cond ((symbol? expr) (make-symbol expr))
          ((number? expr) (make-number expr))
          ((and (boolean? expr) expr) scm-true)
          ((boolean? expr) scm-false)
          ((null? expr) the-empty-list)
          ((pair? expr)
           (make-pair (expr->intern (car expr))
                       (expr->intern (cdr expr))))
          ((eof-object? expr) #f)
          (else (error "; Syntactic error."))))
```

Read a Print

- **reader** – načítá vstup, využijeme `expr->intern` a readeru „metainterpretu“
- **printer** – tiskne výstup, využijeme zabudovaného `display`

;; načti vstupní výraz a převed' jej do interní formy

```
(define scm-read  
  (lambda ()  
    (expr->intern (read))))
```

;; pro daný element vytiskni jeho reprezentaci na obrazovku

```
(define scm-print  
  (lambda (elem)  
    (display elem)))
```

Prostředí jako datová struktura

Vnitřní struktura prostředí:

- **tabulka vazeb**: symbol – element (hodnota navázaná na symbol),
- **ukazatel na předka** (prostředí, které je „výš v hierarchii“).

<i>symbol</i>	<i>element</i>
E_1	F_1
E_2	F_2
\vdots	\vdots
E_k	F_k
\vdots	\vdots

\Rightarrow

$(\langle parent \rangle . ((E_1 . F_1)$
 $(E_2 . F_2)$
 \dots
 $(E_k . F_k)$
 $\dots))$

- potřeba manifestovaných typů – odlišení prostředí a seznamu

Konstruktor prostředí

;; převed' asociační seznam na tabulku prostředí (ve vnitřní reprezentaci)

```
(define assoc->env
  (lambda (l)
    (if (null? l)
        the-empty-list
        (make-pair (make-pair (make-symbol (caar l)) (cdar l))
                    (assoc->env (cdr l))))))
```

;; konstruktor prostředí

```
(define make-env
  (let ((make-physical-env (curry-make-elem 'environment)))
    (lambda (pred table)
      (make-physical-env
       (cons pred table)))))
```

Konstruktor globálního prostředí a detekce typů

```
;; test datového typu „prostředí“
(define scm-env? (curry-scm-type 'environment))

;; konstruktor globálního prostředí
(define make-global-env
  (lambda (alist-table)
    (make-env scm-false (assoc->env alist-table))))

;; testuj, jestli je daný element globální prostředí
(define global?
  (lambda (elem)
    (and (scm-env? elem)
         (equal? scm-false (get-pred elem)))))
```

Základní selektory prostředí

;; vrať předka daného prostředí

```
(define get-pred  
  (lambda (elem)  
    (if (scm-env? elem)  
        (car (get-data elem))  
        (error "; Get-pred: arg. must be an env."))))
```

;; pro dané prostředí vrať tabulku vazeb

```
(define get-table  
  (lambda (elem)  
    (if (scm-env? elem)  
        (cdr (get-data elem))  
        (error "; Get-table: arg. must be an env."))))
```

Hledání vazeb v prostředích

;; hledání vazeb v asociačním seznamu (ve vnitřní reprezentaci)

```
(define scm-assoc  
  (lambda (key alist)  
    ... iterativní prohledávání alist ve vnitřní reprezentaci ...))
```

;; vyhledej vazbu v prostředí *env*, nebo vrať *not-found*

```
(define lookup-env  
  (lambda (env symbol search-nonlocal? not-found)  
    (let ((found (scm-assoc symbol (get-table env))))  
      (cond ((not (equal? found scm-false)) found)  
            ((global? env) not-found)  
            ((not search-nonlocal?) not-found)  
            (else (lookup-env (get-pred env)  
                              symbol #t not-found))))))
```

Primitivní procedury a speciální formy

;; konstruktor primitivních procedur a test datového typu

```
(define make-primitive (curry-make-elem 'primitive))  
(define scm-primitive? (curry-scm-type 'primitive))
```

;; vytváření primitivních procedur pomocí „zabalení metaprocedur“

```
(define wrap-primitive  
  (lambda (proc)  
    (make-primitive  
      (lambda arguments  
        (expr->intern  
          (apply proc (map get-data arguments)))))))
```

;; konstruktor speciálních form test datového typu

```
(define make-specform (curry-make-elem 'specform))  
(define scm-specform? (curry-scm-type 'specform))
```


Uživatelsky definované procedury: konstruktor a selektory

```
(define make-procedure
  (let ((make-physical-procedure
        (curry-make-elem 'procedure)))
    (lambda (env args body)
      (make-physical-procedure (list env args body)))))

(define procedure-environment ...)
(define procedure-arguments ...)
(define procedure-body ...)

(define scm-user-procedure? (curry-scm-type 'procedure))

(define scm-procedure?
  (lambda (elem)
    (or (scm-primitive? elem)
        (scm-user-procedure? elem))))
```

Příklad (další pomocné procedury)

Procedura `map-scm-list->list`

- pracuje jako `map`, akceptuje proceduru a seznam v interní reprezentaci
- výsledkem je metaseznam hodnot

Procedura `scm-list->list`

- akceptuje seznam v interní reprezentaci jako argument
- výsledkem je metaseznam obsahující stejné hodnoty, jako výchozí seznam

Procedura `list->scm-list`

- akceptuje libovolný seznam jako argument
- výsledkem je seznam v interní reprezentaci obsahující stejné hodnoty, jako výchozí seznam

Implementace Eval (začátek...)

;; vyhodnoť element `elem` v daném prostředí `env`

```
(define scm-eval
```

```
  (lambda (elem env)
```

;; element se vyhodnocuje v závislosti na jeho typu

```
(cond
```

;; symboly se vyhodnocují na svou aktuální vazbu

```
((scm-symbol? elem)
```

```
  (let ((binding (lookup-env env elem #t #f))))
```

```
    (if binding
```

```
        (pair-cdr binding)
```

```
        (error "; EVAL: Symbol not bound")))))
```

Implementace Eval (... pokračování ...)

;; element pro vyhodnocení je seznam (pár)

```
((scm-pair? elem)
```

;; nejprve vyhodnotíme první prvek seznamu

```
(let* ((first (pair-car elem))  
      (args (pair-cdr elem))  
      (f (scm-eval first env)))
```

```
(cond
```

;; první prvek je procedura: vyhodnoť argumenty a aplikuj

```
((scm-procedure? f)  
 (scm-apply f (map-scm-list->list  
              (lambda (elem)  
                (scm-eval elem env))  
              args)))
```

←

Implementace Eval (... dokončení)

;; první prvek je speciální forma: aplikuj s nevyhodnocenými argumenty

```
((scm-specform? f)
```

```
(scm-form-apply env f (scm-list->list args))) ←
```

;; první prvek seznamu se vyhodnotil na nepřipustný prvek

```
(error "; EVAL: First element ..."))))
```

;; vše ostatní se vyhodnocuje na sebe sama

```
(else elem))))
```

Z hlediska vyhodnocovacího procesu zbývá implementovat:

- `scm-apply` – metaprocedura realizující aplikaci procedur
- `scm-form-apply` – metaprocedura realizující aplikaci speciálních forem

Vytvoření tabulky vazeb v lokálním prostředí

;; vytvoř tabulku vazeb: formální argument – argument

```
(define make-bindings
  (lambda (formal-args args)
    (cond ((scm-null? formal-args) the-empty-list)
          ((scm-symbol? formal-args)
           (make-pair (make-pair formal-args
                                  (list->scm-list args))
                       the-empty-list))
          (else (make-pair
                   (make-pair (pair-car formal-args)
                              (car args))
                   (make-bindings (pair-cdr formal-args)
                                  (cdr args)))))))
```

Aplikace procedury s explicitním prostředím

;; aplikuj proceduru, předka lokálního prostředí nastav na *env*

```
(define scm-env-apply
  (lambda (proc env args)
    (cond ((scm-primitive? proc)
           (apply (get-data proc) args)) ← metaprocedura
          ((scm-user-procedure? proc)
           (scm-eval (procedure-body proc)
                     (make-env
                      env
                      (make-bindings
                       (procedure-arguments proc)
                       args))))
          (else (error "APPLY: Expected procedure")))))
```

Aplikace procedur s lexikálním rozsahem platnosti

;; standardní aplikace procedury

```
(define scm-apply
  (lambda (proc args)
    (cond ((scm-primitive? proc)
           (scm-env-apply proc #f args))
          ((scm-user-procedure? proc)
           (scm-env-apply
            proc
            (procedure-environment proc)
            args))
          (else (error "APPLY: Expected procedure")))))
```


Aplikace speciálních forem s explicitním prostředím

```
;; aplikace speciální formy
(define scm-form-apply
  (lambda (env form args)
    (cond ((scm-specform? form)
           (apply (get-data form) env args))
          (else (error "APPLY: Expected sp. form")))))
```

Poznámky:

- interpret \times metainterpret
- speciální formy v interpretu = procedury v metainterpretu
- primitivní procedury v interpretu = procedury v metainterpretu
- srovnejte: jak by vypadalo, kdyby byl metajazyk C

Hierarchie prostředí a důsledky neexistence `define`

- rekurze pomocí Y -kombinátorů
- do globálního prostředí nelze během činnosti interpretu zavést nové definice

Hierarchie tří počátečních prostředí

① `toplevel-environment`

- v hierarchii úplně nejvýš (nemá předka)
- obsahuje základní definice (primitivní procedury a spec. formy)

② `midlevel-environment`

- jeho předkem je `toplevel-environment`
- obsahuje definice uživatelských procedur, které jsou k dispozici na počátku běhu interpretu (`map`, `length`, ...)

③ `global-environment` (není v této fázi nezbytné)

- jeho předkem je `midlevel-environment`
- neobsahuje žádné definice

Vytvoření prostředí `toplevel-environment` (začátek...)

;; vytvoř prostředí, které je nejvýš v hierarchii

```
(define scheme-toplevel-env
  (make-global-env
    `(
      ;; implementace speciální formy if
      (if . ,(make-specform
        (lambda (env condition expr . alt-expr)
          (let ((result (scm-eval condition env)))
            (if (equal? result scm-false)
                (if (null? alt-expr)
                    the-undefined-value
                    (scm-eval (car alt-expr) env))
                (scm-eval expr env)))))))
```

Vytvoření prostředí `toplevel-environment` (... pokračování...)

;; speciální formy `and` a `or`

```
(and . , (make-specform ...))
```

```
(or . , (make-specform ...))
```

;; speciální forma `lambda`

```
(lambda . , (make-specform  
              (lambda (env args body)  
                (make-procedure env args body))))
```

;; speciální forma `the-environment` (jeden výraz v těle)

```
(the-environment . , (make-specform  
                     (lambda (env) env)))
```

;; speciální forma `quote`

```
(quote . , (make-specform  
            (lambda (env elem) elem)))
```

Vytvoření prostředí `toplevel-environment` (... pokračování...)

```
;; primitivní procedury pro aritmetiku (převzaté metaprocedury)
(* . ,(wrap-primitive *))
(+ . ,(wrap-primitive +))
:
;; primitivní procedury pro práci s páry
(cons . ,(make-primitive make-pair))
(car . ,(make-primitive pair-car))
(cdr . ,(make-primitive pair-cdr))

;; primitivní procedura negace
(not . ,(make-primitive
          (lambda (elem)
            (if (equal? elem scm-false)
                scm-true
                scm-false)))))
```

Vytvoření prostředí `toplevel-environment` (... pokračování...)

```
;; procedury pro práci s prostředím a procedurami
(environment-parent . ,(make-primitive get-pred))
(procedure-environment . ...
(procedure-arguments . ...
(procedure-body . ...

;; tabulky vazeb prostředí
(environment->list . ,(make-primitive
                        (lambda (elem)
                          (if (equal? elem scm-false)
                              scm-false
                              (get-table elem))))))
```

Vytvoření prostředí `oplevel-environment` (... dokončení)

```
;; primitivní procedura eval dvou argumentů
(eval . ,(make-primitive scm-eval))

;; primitivní procedura apply
(apply . ,(make-primitive
              (lambda (proc . rest)
                (scm-apply proc
                           (apply-collect-arguments rest))))))

;; procedura pro aplikace s explicitním prostředím předka
(env-apply . ,(make-primitive
                  (lambda (proc env . rest)
                    (scm-env-apply proc
                                     env
                                     (apply-collect-arguments rest))))))...
```

Příklad (pomocná procedura pro sestavení argumentů `apply`)

```
;; sestav seznam argumentů pro apply
(define apply-collect-arguments
  (lambda (args)
    (cond ((null? args)
           (error "APPLY: argument missing"))
          ((and (not (null? args)) (null? (cdr args)))
           (scm-list->list (car args)))
          (else (cons (car args)
                       (apply-collect-arguments
                        (cdr args)))))))
```

Je pokryt univerzální přístup:

- pouze seznam argumentů, nebo:
- jednotlivé hodnoty a seznam zbylých argumentů.

Příklad (vytvoření `midlevel-environment`)

```
(define scheme-midlevel-env
  (make-env scheme-toplevel-env
    (assoc->env `(...
      (map . ,(make-procedure
        scheme-toplevel-env
        (expr->intern '(f 1))
        (expr->intern
          '((lambda (y)
            (y y 1))
            (lambda (map 1)
              (if (null? 1)
                '()
                (cons (f (car 1))
                      (map map (cdr 1))))))))))...))))))
```

Implementace cyklu REPL

;; procedura startující vyhodnocovací cyklus REPL

```
(define scm-repl
  (lambda ()
    (let ((glob-env (make-env
                      scheme-midlevel-env
                      the-empty-list)))
      (let loop ()
        (display "]=> ")
        (let ((elem (scm-read)))
          (if (not elem)
              'bye-bye
              (let ((result (scm-eval elem glob-env)))
                (scm-print result)
                (loop))))))))))
```

Rekapitulace ...

O čem to celé bylo?

- 1 o programování v interpretu Scheme
- 2 o programování interpretu Scheme

Procedurální × funkcionální paradigma

- hodnoty proměnných × symboly a jejich vazby
- sekvence přiřazovacích příkazů × postupné aplikace procedur
- cykly × rekurze, iterace, akumulace
- destruktivní × konstruktivní manipulace s hierarchickými daty

Co je exkluzivní?

- procedury vyšších řádů – procedury vytvářející procedury
- program = data v nejčistší možné podobě (v PP2 uvidíte ještě více)