Paradigmata programování II
Zásobníkový model vyhodnocování a implementace call/cc

Vilém Vychodil

Katedra informatiky, Univerzita Palackého v Olomouci

4. května 2006

Implementace `call/cc`

Ukázali jsme

- `call/cc` a jeho aplikace pro řízení výpočtu
  (řízení cyklů, korutiny, nedeterminismus, paralelismus)

Všimli jsme si:

- `call/cc` je silný, ale nebezpečný, nástroj
  (je třeba používat obezřetně, pokud vůbec)

Přirozená otázka:

- Jak těžké je implementovat `call/cc`?
  (odpověď: uvidíme na této přednášce)

Základní myšlenka implementace:

- Vyhodnocovací proces musíme umět uchopit tak, abychom se na něj mohli dívat jako na posloupnost elementárních kroků vyhodnocení
  (povede na vytvoření **iterativní verze procedury „Eval“**)

Rekurzivní vs. iterativní „Eval“

- současná verze „Eval“ je (stromově) rekurzivní
- nejprve vytvoříme interpret jehož „Eval“ bude iterativní
- místo rekurzivního volání „Eval“ budeme používat *iterativní volání* spojené s manipulací s *dodatečnými zásobníky*

```
;; MOTIVACNI PRIKLAD: pocet atomu v seznamu

;; pocet atomu v seznamu (rekurzivni verze)
(define atoms
  (lambda (l)
    (cond ((null? l) 0)
          ((pair? (car l))
           (+ (atoms (car l))
              (atoms (cdr l))))
          (else (+ 1 (atoms (cdr l)))))))
```

```
;; pocet atomu v seznamu (iterativni verze)
(define atoms
  (lambda (l)
    (let iter ((l (list l))
               (accum 0))
      (cond ((null? l) accum)
            ((null? (car l)) (iter (cdr l) accum))
            ((pair? (car l))
             (iter (apply list
                          (caar l)
                          (cdar l)
                          (cdr l))
                   accum))
            (else (iter (cdr l) (+ accum 1)))))))
```

```
(atoms '((a (b)) (c) d))
[((a (b)) (c) d)]   |  0
[(a (b)) ((c) d)]   |  0
[a ((b)) ((c) d)]   |  0
[((b)) ((c) d)]     |  1
[(b) () ((c) d)]    |  1
[b () () ((c) d)]   |  1
[() () ((c) d)]     |  2
[() ((c) d)]        |  2
[((c) d)]           |  2
[(c) (d)]           |  2
[c () (d)]          |  2
[() (d)]            |  3
[(d)]               |  3
[d ()]              |  3
[()]                |  4
[]                  |  4
⟹  4
```

Motivační příklad vyhodnocování

Předpokládáme, že máme **dva zásobníky**:

1. **EXEC** (Execute Stack): zásobník elementů čekajících na vyhodnocení
2. **RSLT** (Result Stack): zásobník výsledků vyhodnocení

```
1. EXEC: (* 20 70)]
   RSLT: ]
2. EXEC: *, #<inspect>, (20 70)]
   RSLT: ]
3. EXEC: #<inspect>, (20 70)]
   RSLT: #<primitive-procedure>]
4. EXEC: 20, 70, #<run>, 2]
   RSLT: #<primitive-procedure>]
5. EXEC: 70, #<run>, 2]
   RSLT: 20, #<primitive-procedure>]
6. EXEC: #<run>, 2]
   RSLT: 70, 20, #<primitive-procedure>]
7. EXEC: ]
   RSLT: 1400]
```

Složitější příklad vyhodnocení (1/2)

```
EXEC: (* 20 (+ 30 40))]
RSLT: ]
  ⇓
EXEC: *, #<inspect>, (20 (+ 30 40))]
RSLT: ]
  ⇓
EXEC: #<inspect>, (20 (+ 30 40))]
RSLT: #<primitive-procedure>]
  ⇓
EXEC: 20, (+ 30 40), #<run>, 2]
RSLT: #<primitive-procedure>]
  ⇓
EXEC: (+ 30 40), #<run>, 2]
RSLT: 20, #<primitive-procedure>]
  ⇓
EXEC: +, #<inspect>, (30 40), #<run>, 2]
RSLT: 20, #<primitive-procedure>]
  ⇓
```

Složitější příklad vyhodnocení (2/2)

```
EXEC: #<inspect>, (30 40), #<run>, 2]
RSLT: #<primitive-procedure>, 20, #<primitive-procedure>]
  ⇓
EXEC: 30, 40, #<run>, 2, #<run>, 2]
RSLT: #<primitive-procedure>, 20, #<primitive-procedure>]
  ⇓
EXEC: 40, #<run>, 2, #<run>, 2]
RSLT: 30, #<primitive-procedure>, 20, #<primitive-proc.>]
  ⇓
EXEC: #<run>, 2, #<run>, 2]
RSLT: 40, 30, #<primitive-procedure>, 20, #<primit.>]
  ⇓
EXEC: #<run>, 2]
RSLT: 70, 20, #<primitive-procedure>]
  ⇓
EXEC: ]
RSLT: 1400]
```

Implementace vyhodnocování pomocí zásobníků

- je implementovaná (téměř tak), jak jsme naznačili v příkladu
- navíc: prostředí, uživatelsky def. procedury, makra, etc.

Zavedeme nové speciální elementy

nazev	symbol s vazbou elementu + popis	
`#<inspect>`	`the-inspect`	rozlišení akcí podle 1. prvku
`#<run>`	`the-run`	aplikace
`#<envrun>`	`the-run`	aplikace s daným prostředím
`#<return>`	`the-return`	vrácení z RSLT zpět na EXEC
`#<return-on-true>`	`the-rotrue`	podmíněné vrácení kvůli `if`
`#<discard>`	`the-discard`	zrušení jednoho prvku z RSLT
`#<define-assign>`	`the-defasgn`	kvůli `define`
`#<set-assign>`	`the-setasgn`	kvůli `set!`

Pravidlo 0 (počátek vyhodnocování elementu v prostředí)

```
EXEC: element, #<env>]
RSLT: ]
```

Pravidlo 1 (symbol se vyhodnotí na svou vazbu v daném prostředí)

```
EXEC: lambda, #<env>]
RSLT: ]
```

$\Downarrow$

```
EXEC: ]
RSLT: #<special-form>]
```

Implementace pravidla 1 v našem interpretu

```
(define rule-eval-symbol?
  (lambda (*EXEC* *RSLT*)
    (and (scm-symbol? (car *EXEC*))
         (scm-env? (cadr *EXEC*)))))

(define rule-eval-symbol
  (lambda ()
    (set! *RSLT*
          (cons
           (let ((binding (lookup-env
                           (cadr *EXEC*)
                           (car *EXEC*) #t #f)))
             (if binding
                 (pair-cdr binding)
                 (error "EVAL: Symbol not bound")))
           *RSLT*))
    (set! *EXEC* (cddr *EXEC*))))
```

Pravidlo 2 (seznam se rozdělí na první prvek, inspektor a zbytek)

```
EXEC: (f arg1 arg2 ... argn), #<env>]
RSLT: ]
  ⇓
EXEC: f, #<env>, #<inspect>, (arg1 arg2 ... argn), #<env>]
RSLT: ]
```

Pravidlo 3 (na vrcholu EXEC je `#<inspect>`)
Je-li na vrcholu RSLT procedura (primitivní/uživatelská):

```
EXEC: #<inspect>, (arg1 ... argn), #<env>]
RSLT: #<procedure>]
  ⇓
EXEC: arg1, #<env> ··· argn, #<env>, #<run>, n, #<env>]
RSLT: #<procedure>]
```

Pravidlo 3 (na vrcholu EXEC je #<inspect>)
Je-li na vrcholu RSLT forma (speciální forma/makro):

```
EXEC: #<inspect>, (arg1 ... argn), #<env>]
RSLT: #<form>]
  ⇓
EXEC: #<run>, n, #<env>]
RSLT: arg1, ..., argn, #<form>]
```

Pravidlo 4 (na vrcholu EXEC je #<run>)
nejsložitější pravidlo, připravuje aplikaci
rozlišuje se primitivní procedura / uživatelská procedura / speciální forma / makro / **manipulující procedura** / **manipulující forma**
(poslední dvě jsou novinky)

Pravidlo 4 (na vrcholu EXEC je #<run>)
Případ: aplikace primitivní procedury (spec. formy – analogické)

EXEC: #<run>, n, #<env>]
RSLT: res1 ··· resn, #<primitive-procedure>]
⇓
EXEC:]
RSLT: výsledek aplikace]

Případ: aplikace uživatelské procedury

EXEC: #<run>, n, #<env>]
RSLT: arg1 ··· argn, #<user-defined-procedure>]
⇓
EXEC: tělo procedury, #<env : nové prostředí s vazbami>]
RSLT:]

Případ: aplikace makra

```
EXEC: #<run>, n, #<env>]
RSLT: arg1 ··· argn, #<macro>]
  ⇓
EXEC: #<run>, n, #<env>, #<return>, #<env>]
RSLT: argn ··· arg1, #<user-defined-procedure>]
```

Vazba na pravidlo #<return>:

```
EXEC: #<return>]
RSLT: element]
  ⇓
EXEC: element]
RSLT: ]
```

Manipulující procedury/formy

- jsou procedury/formy, které mají dva argumenty jimiž jsou samotné zásobníky EXEC a RSLT
- výsledek aktivace manipulující procedury/formy je dvouprvkový seznam `(novy-EXEC novy-RSLT)`
- to jest manipulující procedury/formy mají možnost přímo manipulovat s obahem obou zásobníků a tím pádem *řídit průběh výpočtu* (vymyslel V. Vychodil, Palacký University `:-)`)

Všimněte si: *únikové procedury* lze chápat jako speciální manipulující procedury, které „vyklidí obsah zásobníků a na RSLT zapíšou výsledek“

K tomu abychom mohli dokončit intepret potřebujeme naprogramovat základní formy: `if`, `begin`, `define`, `set!` a některé procedury jako `apply`, `append` (viz zdroják).

Ilustrativní příklad implementace `if`

```
EXEC: (if test expr alt), #<env>]
RSLT: ]
  ⋮
EXEC: #<run>, 3, #<env>]
RSLT: test, expr, alt, #<manipulator-if>]
  ⇓
EXEC: test, #<env>, #<return-on-true>, #<env>]
RSLT: expr, alt]
  ⋮
EXEC: #<return-on-true>, #<env>]
RSLT: cokoliv nebo #f, expr, alt]
  ⇓
EXEC: expr nebo alt, #<env>]
RSLT: ]
```

```
;; specialni forma IF
(if . ,(make-manipulator
        (make-specform
         (lambda (e r)
           (list `(,(car r)
                   ,(caddr e)
                   ,the-rotrue
                   ,@(cddr e))
                 (if (= (get-data (cadr e)) 2)
                     `(,(cadr r)
                       ,the-undefined-value
                       ,@(cdddr r))
                     `(,(cadr r)
                       ,(caddr r)
                       ,@(cddddr r)))))))))
```

Implementace call/cc

- call/cc můžeme udělat jako manipulující proceduru
- *aktuální pokračování* bude rovněž manipulující procedura (aktuální pokračování je procedura předaná receiveru)

```
;; manipulator call/cc (definice v globalnim prostredi)
(call/cc . ,(make-manipulator
             (make-primitive
              (lambda (e r)
                (let ((continuation (make-cc))
                      (receiver (car r)))
                  (list `(,the-run
                          ,(make-number 1)
                          ,@(cddr e))
                        `(,continuation
                          ,receiver
                          ,@(cddr r)))))))))
```

```
;; make-cc: zhmotni aktualni pokracovani
(define make-cc
  (lambda ()
    (let ((saved-EXEC (cdddr *EXEC*))
          (saved-RSLT (cddr *RSLT*)))
      (make-manipulator
       (make-primitive
        (lambda (e r)
          (list saved-EXEC
                (cons (if (= (get-data (cadr e)) 0)
                          the-undefined-value
                          (car r))
                      saved-RSLT)))))))))
```

```
EXEC: (* 20 (call/cc (lambda (f) body))), #<env>]
RSLT: ]
  ⋮
EXEC: (call/cc (lambda (f) body)), #<env>, #<run>, 2,
      #<env>]
RSLT: 20, #<primitive-procedure>]
  ⇓
EXEC: call/cc, #<env>, #<inspect>, ((lambda (f) body)),
      #<env>, #<run>, 2, #<env>]
RSLT: 20, #<primitive-procedure>]
  ⋮
EXEC: #<run>, 1, #<env>, #<run>, 2, #<env>]
RSLT: #<user-defined-proc.>, #<manipulator-call/cc>,
      20, #<primitive-procedure>]

      KONTEXT-EXEC: #<run>, 2, #<env>]
      KONTEXT-RSLT: □, 20, #<primitive-procedure>]
```

```
EXEC: #<run>, 1, #<env>, #<run>, 2, #<env>]
RSLT: #<manipulator-continuation>,
      #<user-defined-procedure>, 20, #<primitive-proc.>]
```

⇓

```
EXEC: body, #<env : prostředí vzniku receiveru>, #<run>, 2, #<env>]
RSLT: 20, #<primitive-procedure>]
```

Pokud body neobsahuje volání únikové funkce,
pak vyhodnocování probíhá normálně.

Předpokládejme, že body je ve tvaru (+ 30 (f 40)):

```
  ⋮
EXEC: (+ 30 (f 40)), #<env>, #<run>, 2, #<env>]
RSLT: 20, #<primitive-procedure>]
  ⋮
EXEC: #<run>, 1, #<env>, #<run>, 2,
      #<env>, #<run>, 2, #<env>]
RSLT: 40, #<manipulator-continuation>, 30,
      #<primitive-procedure>, 20, #<primitive-procedure>]
```

⇓ použití uschovaných zásobníků a dosazení do □

```
EXEC: #<run>, 2, #<env>]
RSLT: 40, 20, #<primitive-procedure>]

EXEC: ]
RSLT: 800]
```