

Proseminar
"Grundlagen höherer Programmiersprachen"
Wintersemester 2002/03
(Kröger, Rauschmayer)

Continuations

Verfasser : Iskrena Avramova
avramova@uni-muenchen.de

GLIEDERUNG

- Einführung
-
- Das "catch and throw"
- Konzept
-
- exceptions
-
- call-with-current-
- continuation (call/cc)
-
- escaping continuations
-
- tree matching
-
- coroutines

EINFÜHRUNG

Der Begriff

Ein Continuation (Fortsetzung) von der Auswertung eines Ausdrucks

Beispiel:

Im folgenden Beispiel sollte der Wert von (f 0) durch 24
geteilt werden und mit 3 multipliziert werden:

(* (/ 24 (f 0)) 3)Da wir den Wert von f nicht ke

Beispiel:

Im folgenden Beispiel ist f die Addition von 4 und 5. Der Zeichen " ^ " zeigt, dass + eine escape Prozedure ist:

□

```
(* 3 (+^ 4 5))  
-> 9
```

□

Der Inhalt vom Control Stack in dem Moment, wo (+^ 4 5) aufgerufen wird, ist <3, * >. Der Aufruf von einer escape Prozedure hat zur Folge, dass der derzeitige Inhalt vom Control Stack "vergessen" wird. (Eigentlich wird er nicht vergessen, sondern ausgetauscht mit dem Stack von höheren Prozeduren (z.B. loop), die auf dem Wert vom Continuation warten.

C A T C H A N D T H R O W

"catch and throw" ist ein einfacher escape - Mechanismus auf funktionaler Basis.

□

Die Kurz - Spezifikation von "catch and throw" ist wie folgt:

```
(catch 'name <code>)  
  (throw 'name <behandlung>)
```

Bedeutung:

```
(*)--> (catch 'bla □  
        code1 □  
        code2 □  
        (throw 'bla <ersatzwert>) □  
        code3)
```

throw

catch

D.h., dass dem Wert in ~~throw~~ **throw** mmern einen Namen (indiese
catch

throw

Implementierung (in Pseudoscheme):

```
(catch 'name (lambda () <code>)) (throw 'name (l
```

Implementierung in Java :

```
try {  
    <code>  
    throw new Name();  
    <code>  
}  
catch(Name e) {  
    <behandlung>  
}
```

Implementierung in SML □

(die erste Zeile ist catch, die zweite throw) :

```
<code> handle Name => <behandlung> □  
raise Name □  
□  
□
```

Wozu sind exceptions gut?

Im folgenden Beispiel wird eine Funktion definiert (ohne Exceptions

```
readFile { open the file; determine its size; alloc
```

Problem:

Was passiert falls mindestens ein der Befehle nicht ausgeführt

```
} else {          errorCode = -2;          }    } else {
```

read the file into memory; close the file; } cat

CALL-WITH-CURRENT-CONTINUATION □ (CALL/CC)

Definition:

Der Operator `call-with-current-continuation` ruft sein Argument □ auf, welches selbst eine Prozedure ist, mit dem Wert "`current □ continuadion`". Und `current continuation` in jedem Moment von □ der Ausführung des Programms ist eine Abstraktion vom Rest □ des Programms.

Beispiele:

```
(+ 1 (call/cc □  
      (lambda (k) □  
        (+ 2 (k 3)))))) □
```

=> 4 □

□

```
(define r #f) □  
(+ 1 (call/cc □  
      (lambda (k) □  
        (set! r k) □  
        (+ 2 (k 3)))))) □
```

=> 4 □

□

```
(r 5) □
```

=> 6 □

□

```
(+ 3 (r 5)) □
```

=> 6 □

□

Auswertung von Funktionen:

-passiv (ohne call/cc):

(define f

(lambda ()

4))

-aktiv (mit call/cc):

(lambda ()

(call/cc

(lambda (return-it)

(return-it 4))))

ESCAPING CONTINUATIONS

Escaping continuations sind der einfachste Gebrauch von call/cc.

**Implementierung einer Prozedur list-product, die die
Elemente einer Liste multipliziert:**

- ohne call/cc:

(define list-product

(lambda (s)

(let recur ((s s))

(if (null? s) 1

(* (car s) (recur (cdr

s))))))

Problem:

Falls ein Element der Liste 0 ist, dann läuft der Prozess sinnlos weiter bis zum Ende der Liste.

Lösung: escaping continuations

```
(define list-product
  (lambda (s)
    (call/cc
      (lambda (exit)
        (let recur ((s s))
          (if (null? s) 1
              (if (= (car s) 0) (exit 0)
                  (* (car s) (recur (cdr s))))))))))
```

Bei Begegnung von einem Element 0 wird das Continuation exit mit 0 aufgerufen und dabei werden weitere Aufrufe von recur vermieden.

TREE MATCHING

Auswertung der Elemente eines Baums:

```
(define show-tree □  
  (lambda(MyTree)□  
    (let loop((ftree (flatten MyTree)))□  
      (cond ((null? ftree) 'skip)□  
            (else ((display (car ftree))□  
                    (loop (cdr ftree)))))))
```

**Bestimmung ob zwei Bäume dieselbe Struktur (engl. □
fringe) (d.h. dieselben Elemente in derselben □
Reihenfolge) haben: □**

□

```
z.B.: (same-fringe? '(1 (2 3)) '((1 2) 3))□  
=> #t□
```

□

```
(same-fringe? '(1 2 3) '(1 (3 2)))□  
=> #f□
```

□

□

—

- Reinfunktionale Implementierung (ohne call/cc):

```
(define same-fringe? □  
  (lambda (tree1 tree2) □  
    (let loop ((ftree1 (flatten tree1)) □  
              (ftree2 (flatten tree2))) □  
      (cond ((and (null? ftree1) (null? ftree2)) #t) □  
            ((or (null? ftree1) (null? ftree2)) #f) □  
            ((eqv? (car ftree1) (car ftree2)) □  
             (loop (cdr ftree1) (cdr ftree2))) □  
            (else #f))))
```

Die Funktion flatten:

```
(define flatten □  
  (lambda (tree) □  
    (cond ((null? tree) '()) □  
          ((pair? (car tree)) □  
           (append (flatten (car tree)) □  
                   (flatten (cdr tree)))) □  
          (else □  
           (cons (car tree) □  
                 (flatten (cdr tree))))))
```

Nachteile:

- Der Algorithmus durchläuft beide Bäume um sie □
"platt" zu machen. Dann geht er wieder durch bis □
er ungleiche Elemente findet. □

- Der Algorithmus verlangt genauso viele cons wie die gesamte Anzahl der Blättern. □

□

- Implementierung mit call/cc:

```
(define tree->generator □  
  (lambda (tree) □  
    (let ((caller '*)) □  
      (letrec □  
        ((generate-leaves □  
          (lambda () □  
            (let loop ((tree tree)) □  
              (cond ((null? tree) 'skip) □  
                    ((pair? tree) □  
                     (loop (car tree)) □  
                     (loop (cdr tree)))) □  
              (else □  
               (call/cc □  
                (lambda (rest-of-tree) □  
                  (set! generate-leaves □  
                    (lambda () □  
                      (rest-of-tree 'resume))) □  
                  (caller tree)))))) □  
              (caller '())))) □  
          (lambda () □  
            (call/cc □  
             (lambda (k) □  
               (set! caller k) □  
               (generate-leaves)))))) □  
          (caller '())))) □  
    (lambda () □  
      (call/cc □  
       (lambda (k) □  
         (set! caller k) □  
         (generate-leaves)))))) □
```

□

Die Prozedure `same-fringe?` weist jedes ihrer Argumente einem Generator zu und dann werden beide Generatoren abwechselnd aufgerufen. Sobald sie zwei ungleiche Elemente findet, gibt sie Fehlermeldung aus.

```
(define same-fringe?  
  (lambda (tree1 tree2)  
    (let ((gen1 (tree->generator tree1))  
          (gen2 (tree->generator tree2)))  
      (let loop ()  
        (let ((leaf1 (gen1))  
              (leaf2 (gen2)))  
          (if (eqv? leaf1 leaf2)  
              (if (null? leaf1) #t (loop))  
              #f))))))
```

C O R O U T I N E S

Definition:

Coroutines sind einstellige Prozeduren, die sich gegenseitig aufrufen und Ergebnisse austauschen.

```
(coroutine (lambda (v) <body>))
```

<body> enthält die zweistelige Prozedur `resume`

`(resume <co> <val>)`

wobei `<val>` das Ergebnis ist und es wird der Coroutine `<co>` zugeschickt.

Beispiel:

Wir definieren zwei Coroutines `foo` and `goo`. Jede Coroutine druckt ihren Name aus und den derzeitigen Wert von ihrem Parameter bevor die andere mit dem neuen Wert anfängt.

```
(define foo
  (coroutine
    (lambda (m)
      (letrec ([loop
                (lambda ()
                  (printf "foo: ~a~n" m)
                  (if (<= m 100)
                      (begin
                        (set! m (resume goo (+ m 2)))
                        (loop)))))]
              (loop))))))

```

```
(define goo□  
  (coroutine□  
    (lambda (n)□  
      (letrec ([loop□  
                (lambda ()□  
                  (printf "goo: ~a~n" n)□  
                  (set! n (resume foo (- n 1)))□  
                  (loop))]□  
                (loop))))□  
      (loop)))))
```