Ludwig-Maximilians-Universität München
Institut für Informatik

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 E im Kontext G stellt die ganze Zukunft der Berechnung, die auf dem Wert von E wartet, dar.

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 kennen, können wir folgende Vermutungen machen:

1 Fall: **Sei (f 0) z.B. 4**
⇨ das Ergebnis ist 18

2 Fall: **f ist nicht definiert bei 0 ⇨ Fehlermeldung**
⇨ Prozessabbruch

3 Fall: **f ist nicht definiert bei 0 ⇨ unendliche Schleife**

```
z.B.:   (define f
            (lambda (n)
                (if (zero? n) (f n) n)))
```

Hier ist der then-Fall wieder (f n).
Der Prozess terminiert nicht.

4 Fall: **Sei f ein Continuation, d.h. dass (f 0) ausgeführt wird und das Ergebnis als Ergebnis vom ganzen Prozess geliefert wird.**
Sei (f 0) z.B. 4
⇨ das Ergebnis ist 4

Da das Ergebnis von (f 0) den restlichen Berechnungen entgeht, sagen wir auch, dass die Continuations entgehende (escape) Prozeduren sind.

Beispiel:

Im folgenden Beispiel ist f die Addition von 4 und 5. Der Zeichen " ^ " zeigt, dass + eine escape Prozedur 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 Prozedur hat zur Folge, dass der derzeitige Ihalt 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)
```

D.h., dass dem Wert in den Klammern ein Name (in diesem Fall der Name 'bla) zugewiesen wird. `throw` verlässt genau die `catch` - Klammern, deren Namen es angibt. Zusätzlich gibt `throw` an, welchen Ersatzwert der `catch` - Block zurückliefern soll, da dieser nicht komplett ausgeführt werden kann, weil `throw` ihn frühzeitig verlässt.

Implementierung (in Pseudoscheme):

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

Beispiel in Pseudo-Scheme :

```
(define (list-length lst)
  (catch 'exit
    (letrec ((list-length1
              (lambda (lst)
                (cond ((null? lst) 0)
                  ((pair? lst) (+ 1 (list-length1 (cdr lst))))
                  (else (throw 'exit 'improper-list))))))
      (list-length1 lst))))
```

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
```

OUTGOING-ONLY CONTINUATIONS
(E X C E P T I O N S)

Definition :

Ein Exception ist eine Ausnahme (meistens ein Fehler), die bei der Ausführung vom Programm auftritt und den normalen Lauf der Befehle unterbricht.

Wozu sind exceptions gut?

Im folgenden Beispiel wird eine Funktion definiert (ohne Exceptions), die eine komplette Datei (file) im Speicher (memory) liest. (im Pseudocode) :

```
readFile {
   open the file;
   determine its size;
   allocate that much memory;
   read the file into memory;
   close the file;
}
```

Problem:

Was passiert falls mindestens ein der Befehle nicht ausgeführt werden kann?

Lösung ohne Exceptions:

```
errorCodeType readFile {
   initialize errorCode = 0;
   open the file;
   if (theFileIsOpen) {
      determine the length of the file;
      if (gotTheFileLength) {
         allocate that much memory;
         if (gotEnoughMemory) {
            read the file into memory;
            if (readFailed) {
               errorCode = -1;
            }
```

```
            } else {
               errorCode = -2;
            }
         } else {
            errorCode = -3;
         }
         close the file;
         if (theFileDidntClose && errorCode == 0) {
            errorCode = -4;
         } else {
            errorCode = errorCode and -4;
         }
      } else {
         errorCode = -5;
      }
      return errorCode;
   }
```

Hier sind aber die Hauptbefehle unter den if - Schleifen verloren. Da man die logische Abfolge der Befehle schwer erkennen kann, ist die Korrektheit der Ergebnisse nicht mehr sicher.

Lösung mit Exceptions (Pseudo-Java) :

```
   readFile {
      try {
         open the file;
         determine its size;
         allocate that much memory;
```

```
        read the file into memory;
        close the file;
    } catch (fileOpenFailed) {
        doSomething;
    } catch (sizeDeterminationFailed) {
        doSomething;
    } catch (memoryAllocationFailed) {
        doSomething;
    } catch (readFailed) {
        doSomething;
    } catch (fileCloseFailed) {
        doSomething;
    }
}
```

Man macht sich zwar wieder die Mühe, die Ausnahmen und die Ausnahmebehandler zu definieren, aber sie sind getrennt von den Hauptbefehlen.

Scheme unterstützt keine "catch and throw" - Methoden und keine Exceptions. Um escape Prozeduren in Scheme zu definieren, benutzt man den Operator call-with-current-continuation. Der Name des Operators ist ziemlich lang. Deswegen wird stattdessen call/cc benutzt.

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 continuation". 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
```

Im obigen Beispiel ist der Rest des Programms das folgende "Programm mit einem leeren Platz":

```
( +  1  [  ] )
```

wobei [ ] der leere Platz ist. Mit anderen Worten, dieses Continuation ist ein Programm, das 1 zu "etwas" addiert , wobei "etwas" an der Stelle des leeren Platzes kommt.

Das Argument von call/cc ist die Prozedur

```
(lambda (k)
  (+ 2 (k 3)))
```

Das Body der Prozedur wendet die Continuation (die jetzt an k gebunden ist) auf dem Argument 3 an. Hier verlässt das

Continuation seine eigene Berechnung und vertauscht sie mit dem Rest des Programms, der in k gespeichert worden ist. D.h., dass die Addition mit 2 nicht ausgeführt wird und das Argument 3 von k direkt dem "Programm mit einem leeren Platz" zugeschickt wird:

```
(+ 1 [ ] )
     ↑
     3
```

Das Programm läuft jetzt einfach:

```
(+ 1 3 )
 -> 4
```

Die obige Prozedur ist ein Continuation, das aus einer Berechnung rausspringt (hier die Addition mit 2). Scheme Continuations können auch zu einem vorher-verlassenen Kontext zurückspringen und ihn mehrmals aufrufen. Zum Beispiel:

```
(define r #f)
(+ 1 (call/cc
      (lambda (k)
        (set! r k)
        (+ 2 (k 3)))))
=> 4
```

Das ist dasselbe Ergebnis, aber diesmal ist das Continuation k in einer globalen Variablen r gespeichert. Und jedesmal wenn wir r von einer Zahl aufrufen, werden wir die Zahl + 1 erhalten:

```
(r 5)
=>  6
```

r verlässt auch seine eigene Berechnung:

```
(+ 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))))
```

E S C A P I N G C O N T I N U A T I O N S

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.

T R E E M A T C H I N G

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 Blätter-Struktur (engl. fringe) (d.h. dieselben Blätter 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 genausoviele cons wie die gesamte Anzahl der Blättern.

- <u>Implementierung mit call/cc:</u>

Mit call/cc kann man das Problem ohne sinnloses Durchlaufen und ohne cons lösen:

```
(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)))))))))
```

Der Generator durchläuft den Baum und findet alle Blätter von links nach rechts in der Reihenfolge, in der sie im Baum vorkommen.

Wenn tree->generator aufgerufen wird, wird das Continuation von seinem Aufruf im caller gespeichert, so dass er wissen wird wem er das später gefundene Blatt schicken soll. Dann ruft er eine Prozedur generate-leaves auf, die durch ein loop den Baum von links nach rechts durchläuft.Wenn er ein Blatt findet, wird er den caller benutzen, um das Blatt als Ergebnis des Generators zurückzuliefern. Der Rest vom loop wird in die generate-leaves Variable gespeichert. Und das nächste Mal, wenn der Generator aufgerufen wird, wird der loop genau ab dort weiterlaufen, wo er aufgehört hat.

Die Prozedur 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 und goo. Jede □ Coroutine gibt ihren Namen 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)))))□
```