Fakultät mit CPS

Ohne CPS:

```
fun fac n = if n<=1 then 1 else n*fac(n-1)
```

Im CPS:

```
fun fac1 (n,k) = if n<=1 then k 1
                 else fac1 (n-1,fn x => k (n*x))
val fac1 = fn : int * (int -> 'a) -> 'a

fac1 (3,fn x=> x);
val it = 6 : int
```

Vorteile der CPS

- Optimierung des Speicher-Verhaltens: Jeder Aufruf, insbesondere end-rekursive Aufrufe, liefern stets den Wert der aufrufenden Funktion:
  - ▷ Transformation der rekursiven Funktionen in end-rekursive Funktionen
  - ▷ Compiler-Optimierungen: Der SML-Compiler benutzt CPS als Zwischendarstellung für Compilierung und Optimierungen $\Longrightarrow$ unnötige Rücksprünge werden eliminiert
- Erhöhung der Ausdrucksstärke: Explizitheit der Kontrolflusses
  - ▷ benutzer-definierte Kontrollstrukturen durch explizite Modellierung des Kontrolflusses: z.B. Threads, Korutinen, Ausnahmen

CPS und Effizienz

Im CPS sind die Rückgaben aus Funktionen unnötig.

$\Longrightarrow$ Der Aufruf einer Continuation (Die Aktivierung einer Continuation) braucht nicht die Kontrolle an den Aufrufenden zurückzugeben.

$\Longrightarrow$ Programme in CPS können effizient implementiert werden.

Implizite Continuations

- Soweit haben wir Continuations explizit konstruiert und durchgereicht.
- Jede Berechnung eines Ausdruckes in SML hat eine implizite Continuation:

  die aktuelle Continuation (*current continuation*)

Dies ist die Funktion, die die Zukunft der Auswertung dieses Ausdruckes repräsentiert, d.h. eine Abstraktion dessen, was das System mit dem Wert des Ausdruckes machen wird.

Implizite Continuations

Betrachten wir den Ausdruck:

```
1 + 2 * 3
```

Die impliziten Continuations für jeden Teilausdruck sind unten dargestellt:

Wert	Aktuelle Continuation
`1 + 2 * 3`	`k` (abhängig vom Kontext der Auswertung)
`1`	`fn x => k (x + 2 * 3)`
`2`	`fn x => k (1 + x * 3)`
`3`	`fn x => k (1 + 2 * x)`
`2 * 3`	`fn x => k (1 + x)`

Implizite Continuations benutzen

In SML of New Jersey (wie auch Scheme, Python, u.a.) ist die aktuelle Continuation ein "first class value":

- Man kann auf die aktuelle Continuation zugreifen und sie als "first class value" manipulieren
- Man kann Continuations aktivieren

Implizite Continuations benutzen

Das Modul `SMLofNJ` stellt einen Typ und Operationen für Continuations zur Verfügung:

```
type 'a cont
val callcc : ('a cont -> 'a) -> 'a
val throw : 'a cont -> 'a -> 'b
```

- Werte vom Typ `'a cont` repräsentieren Fortsetzungen von Berechnungen die Werte vom Typ `'a` zurückliefern.
- Die aktuelle Continuation wird mit Hilfe von callcc (*call with current continuation*) explizit verfügbar.
- Eine Continuations kann mit Hilfe von throw aktiviert werden.

Implizite Continuations benutzen

- `callcc (fn k => e)` macht die Fortsetzung der Berechnung, die `e` benutzt (die aktuelle Continuation), innerhalb des Ausdrucks `e` selbst verfügbar.
- `throw k a` aktiviert die Continuation `k` mit dem Wert `a`

Implizite Continuations benutzen

```
1 + callcc (fn k => e)
```

- Wenn `e` die Continuation `k` nicht aktiviert:

$\Longrightarrow$ `callcc (fn k => e)` liefert den Wert von `e`

$\Longrightarrow$`1 +` den Wert von `e` wird zurückgeliefert.

- Wenn bei der Auswertung von `e` die Continuation `k` via `throw k e'` aktiviert:

$\Longrightarrow$ die Berechnung fortgesetzt als hätte `callcc (fn k => e)` den Wert von `e'` zurückgeliefert

$\Longrightarrow$`1 +` den Wert von `e'` zurückgeliefert.

Continuations

```
1 + callcc (fn k => 2);
val it = 3 : int
1 + callcc (fn k => throw k 3);
val it = 4 : int
```

```
fun plist l = case l of nil => 1
                      | 0::_ => 0
                      | h::r => h * (plist r)
```

`plist [1,2,3,4,5,0,6,7,8]` berechnet `1*(2*(3*(4*(5*0))))`.

Besser: mit Continuations

```
fun plist1 l = callcc ( fn k =>
                        let
                          fun p l =
                            case l of nil => 1
                                    | 0::_ => throw k 0
                                    | h::r => h * (p r)
                        in
                          p l
                        end)
```

`plist1 [1,2,3,4,5,0,6,7,8]` liefert direkt 0.

Continuations-Anwendung: Coroutinen

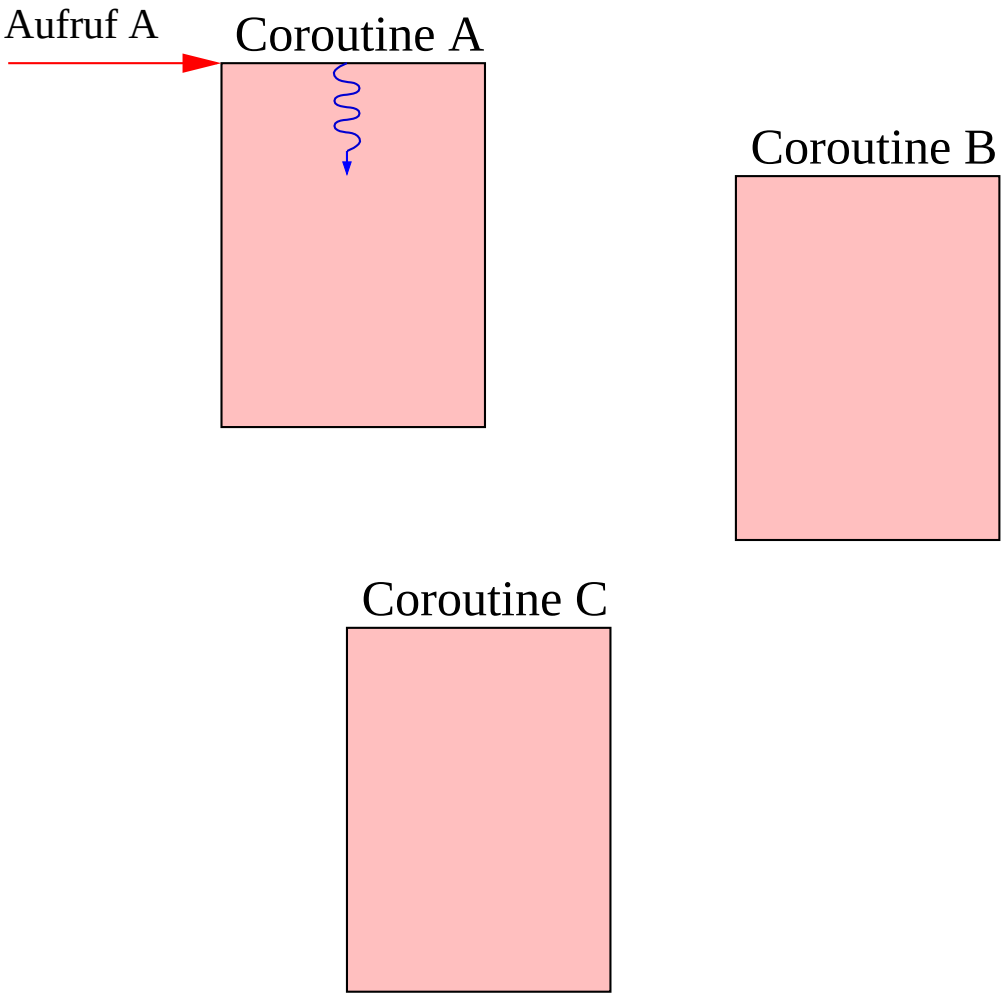
Coroutine = Funktion, die, nach dem sie einen Wert zurückliefern, in dem zuletzt verlassenen Zustand fortgesetzt werden kann.

- Der erste Startpunkt der Coroutinen ist der Anfangspunkt der Coroutine
- Nach einer Rückgabe setzt die Berechnung bei einem neuen Aufruf nach dem Rückgabepunkt fort.

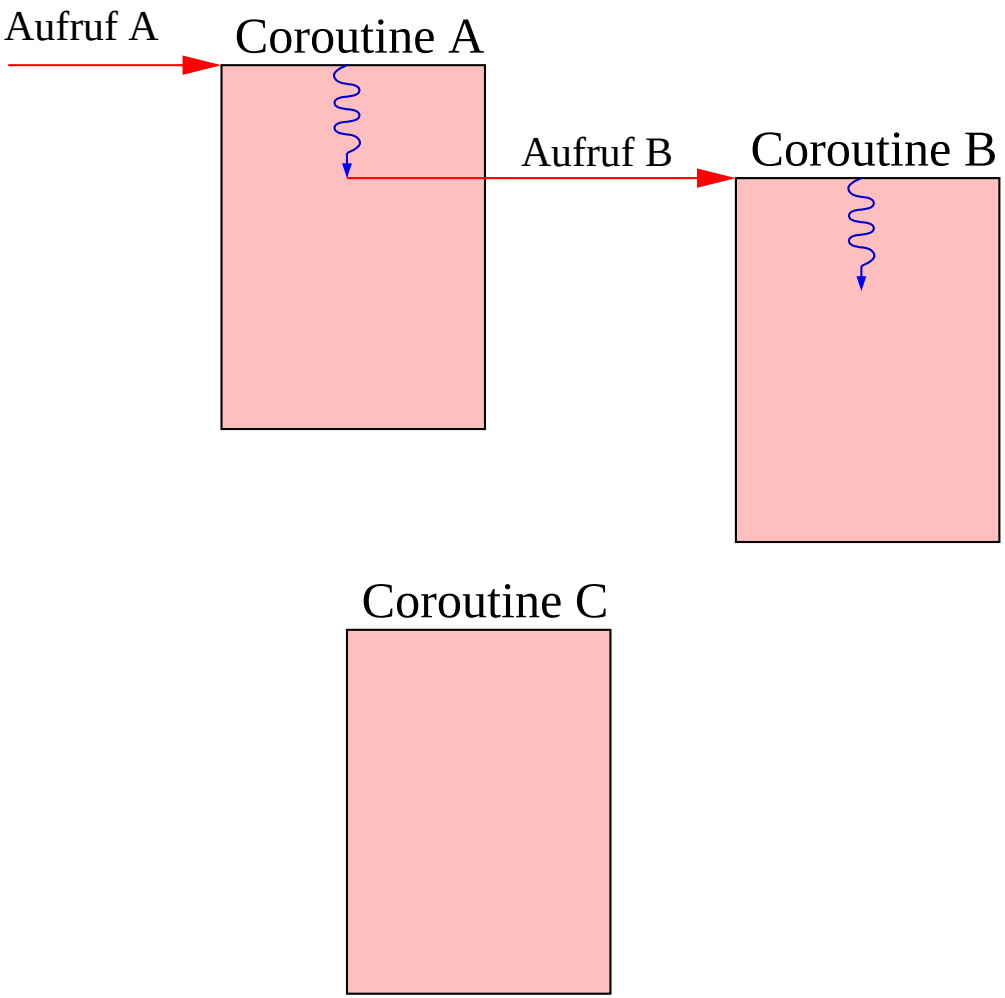
$\Longrightarrow$ Verallgemeinerung von Funktionen:

- mehrere Eingangspunkte
- mehrere Rückgaben aus einer einzigen Funktionsinstanz

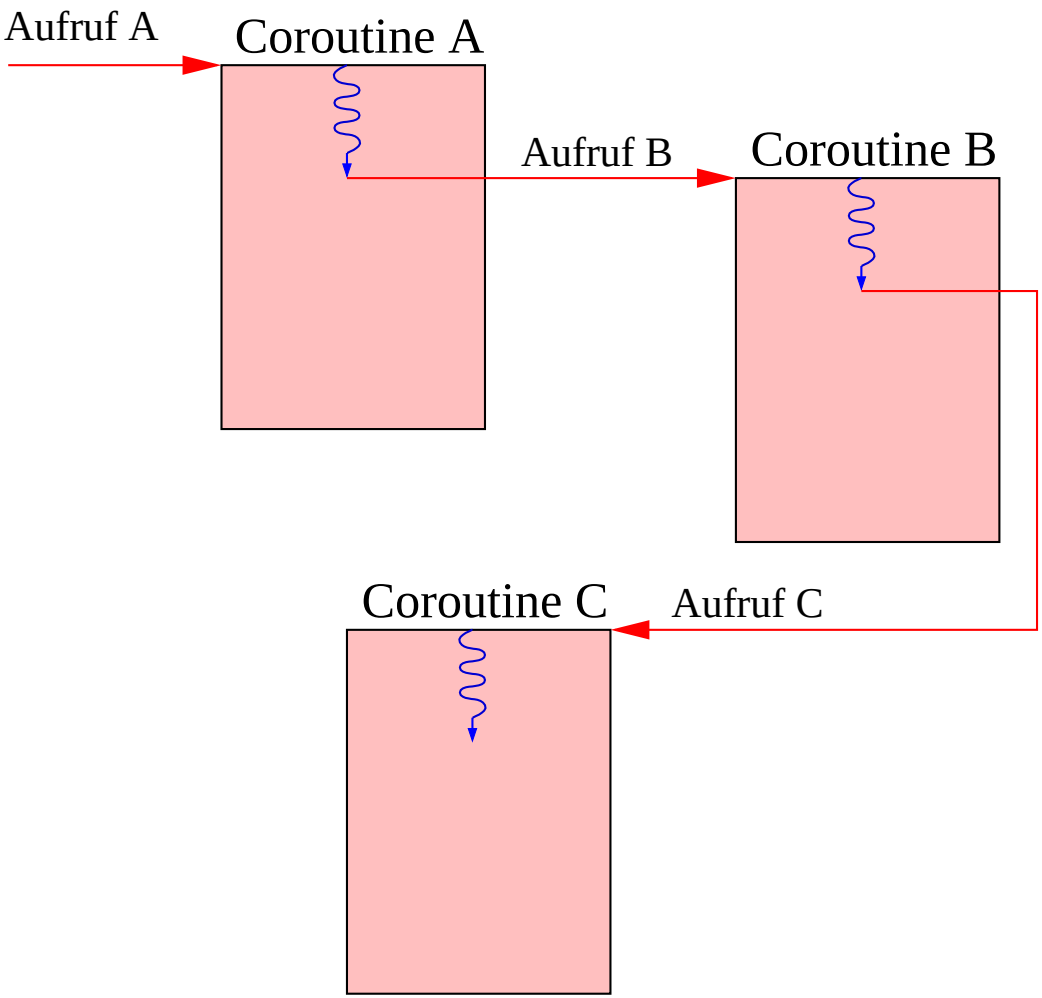
Coroutinen



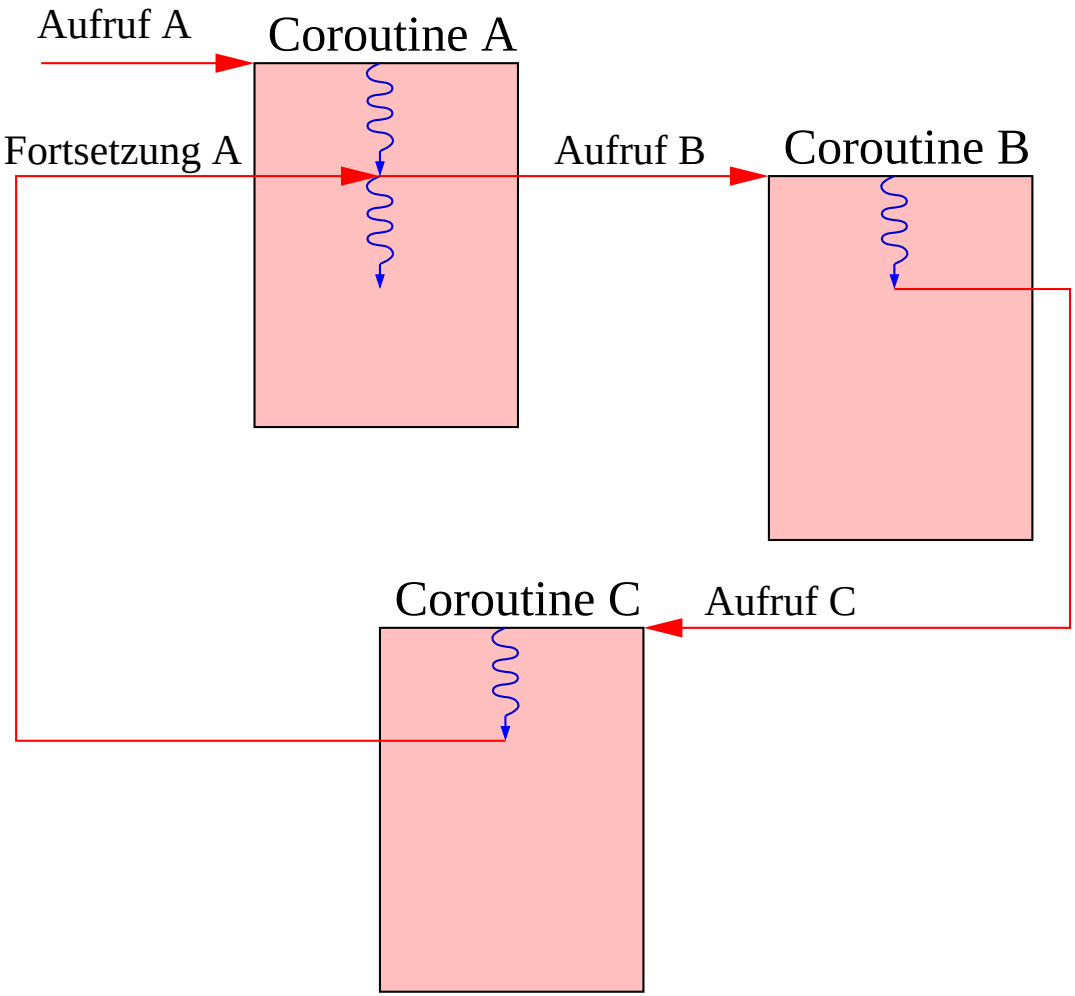
Coroutinen



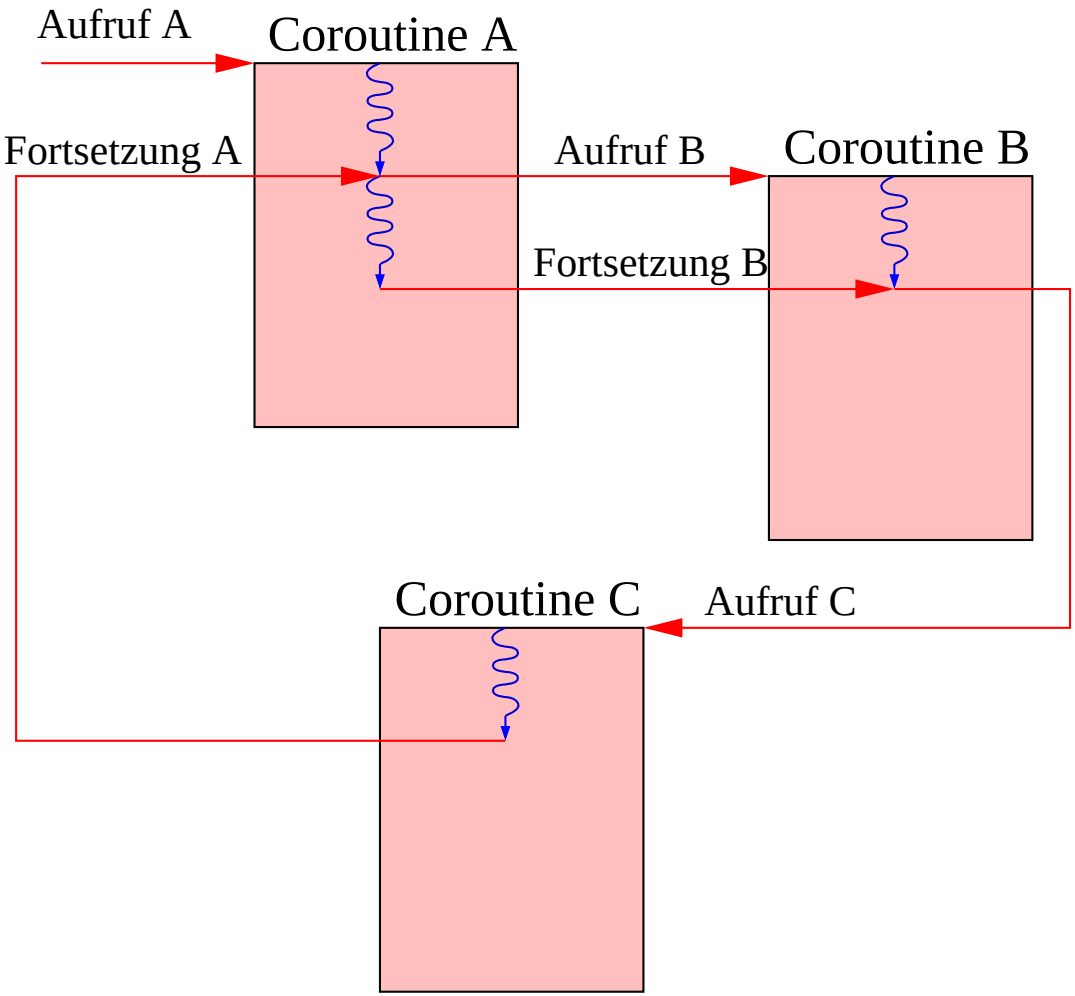
Coroutinen


Aufruf A
Coroutine A
Aufruf B
Coroutine B
Coroutine C
Aufruf C

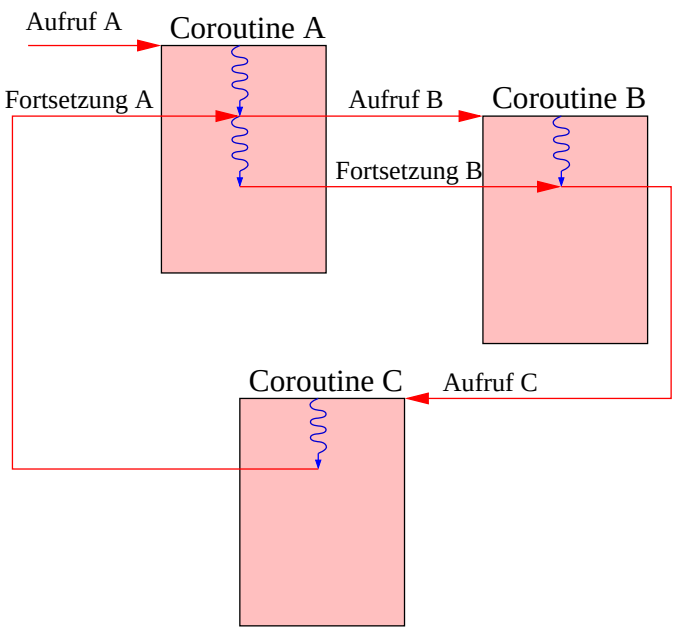
Coroutinen


Aufruf A
Coroutine A
Fortsetzung A
Aufruf B
Coroutine B
Coroutine C
Aufruf C

Coroutinen


Aufruf A
Coroutine A
Fortsetzung A
Aufruf B
Coroutine B
Fortsetzung B
Coroutine C
Aufruf C

Coroutinen



Statt Subordination der Aufgerufenen gegenüber der Aufrufenden

$\Longrightarrow$ Koordination $\Longrightarrow$ Coroutinen

Die Erzeuger-Verbraucher-Architektur mit Coroutinen

Die Ergeuger-Verbraucher-Architektur:

- Der Erzeuger stellt eine Resource zur Verfügung
- Der Verbraucher benutzt die Resource
- Die Kontrolle liegt alternativ beim Erzeuger bzw. Verbraucher

Die Erzeuger-Verbraucher-Architektur mit Coroutinen

```
val buf = ref 0

fun produce (n,consumer:state) =
  (buf := n; produce (n+1,resume consumer))

fun consume (producer:state) =
  (print (Int.toString (!buf));consume (resume producer))
```

- `resume`-Aufrufe implementieren die Übergabe der Kontrolle zwischen dem Erzeuger und dem Verbraucher.
- Bei der Übergabe der Kontrolle muss der aktuelle Zustand des Aufrufenden mit übergeben werden.
- Der Zustand einer Coroutine ist eine Continuation

Die Erzeuger-Verbraucher-Architektur mit Coroutinen

Implementierung der Kontrollübergabe:

```
val buf = ref 0

datatype state = S of state cont

fun resume (S k) = callcc (fn k1 => throw k (S k1))

fun produce (n,consumer:state) =
  (buf := n; produce (n+1,resume consumer))

fun consume (producer:state) =
  (print (Int.toString (!buf));consume (resume producer))
```

Die Erzeuger-Verbraucher-Architektur mit Coroutinen

Äquivalent dazu:

```
val buf = ref 0

datatype state = S of state cont

fun produce (n,S cons) =
  (buf := n;
   produce (n+1,callcc (fn k => throw cons (S k))))

fun consume (S prod) =
  (print (Int.toString (!buf));
   consume(callcc (fn k => throw prod (S k))))
```

Die Erzeuger-Verbraucher-Architektur mit Coroutinen

Anfang:

```
val buf = ref 0

datatype state = S of state cont

fun produce (n,S cons) =
  (buf := n;
   produce (n+1,callcc (fn k => throw cons (S k))))

fun consume (S prod) =
  (print (Int.toString (!buf));
   consume(callcc (fn k => throw prod (S k))))

fun run () = consume (callcc (fn k => produce (0, S k)))
```

Continuations-Anwendung: Threads

Mit zunehmender Anzahl Coroutinen in einem Programm wird die explizite Übergabe der Kontrolle mühsam und unübersichtlich.

Besser: Threads

- flexibler und modularer Ansatz des Kontrollflusses
- asynchrone Events können auch behandelt werden

Eine Coroutine Scheduler koordiniert die verschiedenen Threads.

Die Threads sind Coroutinen des Schedulers.

Continuations-Anwendung: Threads

Scheduler-Funktionalität:

- verwaltet eine Schlange `ready` von Threads
- wenn aufgerufen (via `dispatch`), wählt einen Thread und setzt diesen fort
- Ein Thread ist eine `unit cont`

Definition eines Typ-Synonyms

```
type thread = unit cont
```

Continuations-Anwendung: Threads

Scheduler-Funktionalität:

```
exception NoMoreThreads
type thread = unit cont

val ready : thread Queue.queue = Queue.mkQueue ()

fun dispatch () =
  throw (Queue.dequeue ready) ()
  handle Queue.Dequeue => raise NoMoreThreads
```

Continuations-Anwendung: Threads

Threads-Funktionalität:

- `fork : (unit -> unit) -> unit`

  `fork f` erzeugt einen neuen Thread, der die Funktion f ausführt. Der Aufrufende Thread wird suspendiert.

- `yield : unit -> unit`

  `yield ()` übergibt die Kontrolle an den Scheduler

- `exit : unit -> 'a`

  `exit ()` beendet den Aufrufenden Thread

Continuations-Anwendung: Threads

```
fun enqueue t = Queue.enqueue (ready,t)

fun exit () = dispatch ()

fun fork f =
  callcc (fn parentCont => (enqueue parentCont; f(); exit()))

fun yield () =
  callcc (fn cont => (enqueue cont; dispatch ()))
```

Erzeuger-Verbraucher-Threads

```
val buffer = ref 0

fun producer () =
  (buffer := !buffer + 1;yield();producer ())

fun consumer () =
  (print (Int.toString (!buffer));yield();consumer())

fun run () =
  (init ();fork consumer;producer ())

fun run2 () =
  (init ();fork consumer; fork producer; producer ())
```

Continuations: Fazit

- Da die aktuelle Continuation "first class" ist, lassen sich grundlegende Konstrukte der Nebenläufigkeit leicht implementieren
- Andere Konstrukte lassen sich ebenso mit Hilfe von "first class" Continuations definieren, z.B.: Ausnahmen, Lösungs-Generatoren, Multi-Agent-Programmierung, Iteratoren, Backtracking, andere benutzer-definierte Kontrollstrukturen ($\longrightarrow$ Übung)

4.13 Das Modul-System von SML

4.13.1 Strukturen

In SML heissen die Module Strukturen:

```
structure Pairs =
  struct
    type 'a Pair = 'a * 'a
    fun Pair(a,b) = (a,b)
    fun first (a,b) = a
    fun second (a,b) = b
  end
```

Strukturen

Auf die Eingabe einer Struktur antwortet der Compiler mit dem Typ der Struktur, einer Signatur:

```
structure Pairs :
  sig
    type 'a Pair = 'a * 'a
    val Pair : 'a * 'b -> 'a * 'b
    val first : 'a * 'b -> 'a
    val second : 'a * 'b -> 'b
  end
```

Strukturen

Die Definitionen innerhalb der Struktur sind außerhalb zunächst nicht sichtbar:

```
- first;
stdIn:41.1-41.6 Error: unbound variable or constructor: first
```

Über ihren Namen kann man in das Innere einer Struktur hineinsehen:

```
- Pairs.first;
val it = fn : 'a * 'b -> 'a
```

Strukturen

So kann man z.B. Funktionen für verschiedene Typen mit dem gleichen Namen definieren:

```
structure Triples =
  struct
    datatype 'a Triple = Triple of 'a * 'a * 'a
    fun first (Triple abc) = #1 abc
    fun second (Triple abc) = #2 abc
    fun third (Triple abc) = #3 abc
  end

- Triples.first;
val it = fn : 'a Triples.Triple -> 'a
```