Programmiersprachen

Alexandru Berlea

Institut für Informatik
TU München

Wintersemester 2006/2007

Inferenz-Regeln für Referenz-Typen

Ref:
$$\frac{e : t}{\texttt{ref}\ e : \texttt{ref}\ t}$$

Deref:
$$\frac{e : \texttt{ref}\ t}{\texttt{!}e : t}$$

Assign:
$$\frac{e_1 : \texttt{ref}\ t \qquad e_2 : t}{e_1\ \texttt{:=}\ e_2 : \texttt{unit}}$$

Diese Regeln vertragen sich aber nicht mit Polymorphie!

Polymorphische Referenzen

- Obigen Inferenz-Regeln akzeptieren:

```
let
  val id = fn x => x
  val fp = ref id
  val (x,y) = (!fp 1,!fp true)
in
  (x,y)
end
```

... mit dem (*verallgemeinerten*) polymorphischen Typ $\forall$'a.('a->'a) `ref` für `fp` und frischen neuen Variablen `'a1 = int` bzw. `'a2 =bool` bei jeder Anwendung.

Polymorphische Referenzen

- Die obigen Inferenz-Regeln akzeptieren auch:

```
let
  val id = fn x => x
  val fp = ref id
  val (x,y) = (!fp 1,!fp true)
  val () = fp := !not
  val z = !fp 2
in
  (x,y,z)
end
```

  $\Longrightarrow$ würde Typ-Fehler bei der Laufzeit erzeugen!

- Obiger Code sollte äquivalent sein zu:

```
(!(ref (fn x => x)) 1, !(ref (fn x => x)) true,
 !(ref not) 2)
```

  welcher korrekterweise nicht getypt werden kann.

- Ist aber nicht äquivalent. Problem:
  - ursprünglich: eine Referenz wird alloziert
  - modifizierte Version: drei Referenzen

Polymorphische Referenzen

- Lösung = **Value Restriction**: im Scope einer Deklaration `val id = e` wird der Typ von `id` generalisiert (jedes Auftreten eine anderen Instanz), nur wenn `e` ein **syntaktischer Wert** `v` ist:

```
v ::= Konstante | Bezeichner | Konstruktor v | fn x => Ausdruck
```

- $\Longrightarrow$ Folgendes wird abgewiesen:

```
let
  val id = fn x => x
  val fp = ref id
  val (x,y) = (!fp 1,!fp true)
  val () = fp := !not
  val z = !fp 2
in (x,y,z) end
```

- Aus dem selben Grund werden top-level Deklarationen `val id = e` verboten, wenn `e` kein syntaktischer Wert ist.

Continuations

Rekursionsarten

Je nach Art der rekursiven Aufrufe unterscheiden wir folgende Arten von Rekursion:

- **End-Rekursion** (lineare Rekursion):
  - Bei der Funktionsauswertung gibt es nur einen rekursiven Aufruf.
  - Dieser ist gleichzeitig der Rückgabewert.

```
fun fac1 (n,acc) = if n=1 then acc
                   else fac1(n-1,n*acc)

fun loop x = if x<2 then x
             else if x mod 2 = 0 then loop(x div 2)
                  else loop(3*x+1)
```

Rekursionsarten

- **Repetitive Rekursion:** nur einen rekursiven Aufruf, der aber nicht der Rückgabewert ist.

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

- **Baumartige (kaskadenartige) Rekursion:** mehrere, nicht verschachtelte rekursive Aufrufe.

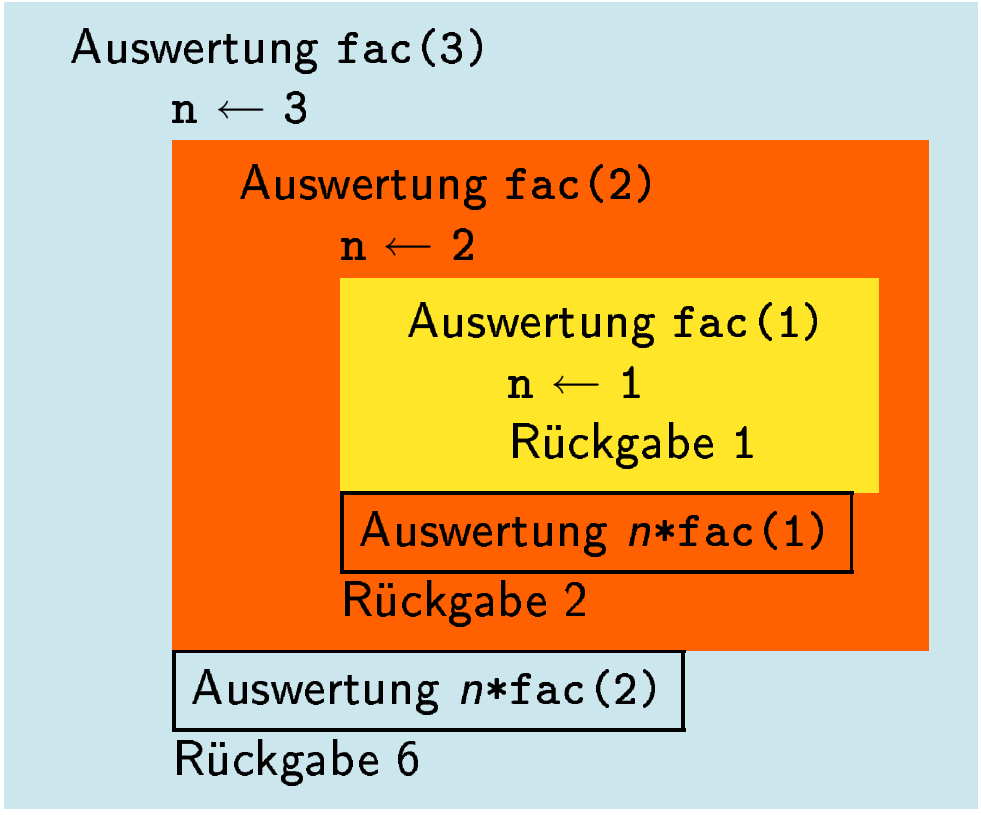
```
fun fib n = if n=0 then 1
            else if n=2 then 1
                 else fib(n-1)+fib(n-2)
```

- **Wilde Rekursion:** verschachtelte rekursive Aufrufe

```
fun f n = if n<2 then n else f(f(n div 2))
```

Speicherverhalten der repetitiv-rekursiven Fkt.

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

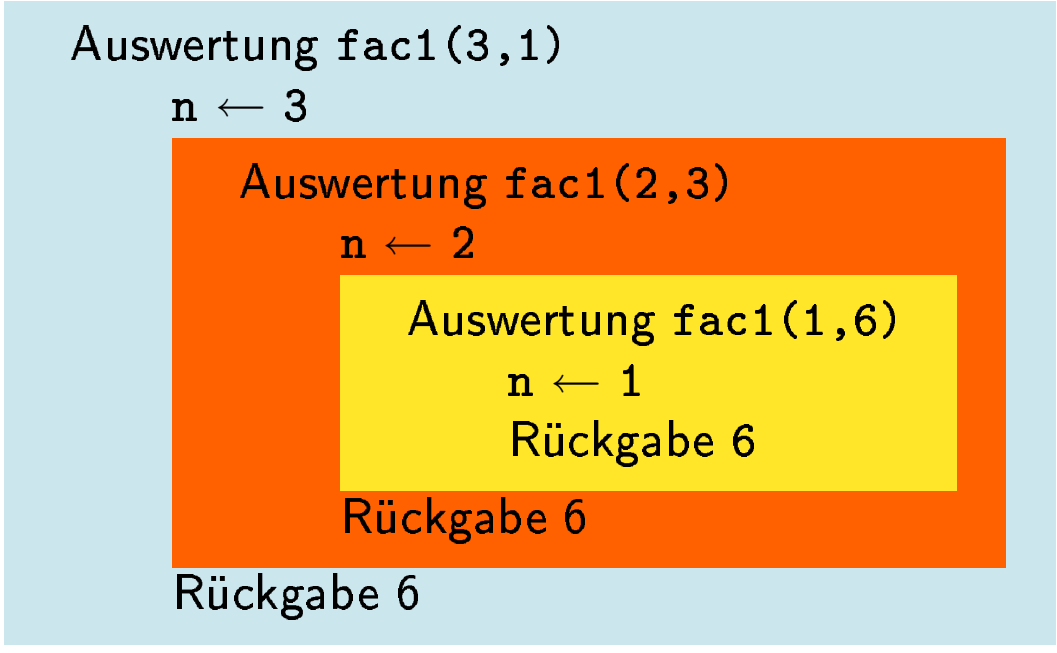


Bei jedem neuen Funktionsaufruf muss der aktuelle Aufruf seine lokale Variablen speichern, um diese nach dem Aufruf verwenden zu können.

$\Longrightarrow$ Speicherverbrauch wächst mit Anzahl geschachtelter Aufrufe.

Speicherverhalten der tail-rekursiven Funktionen

```
fun fac1 (n,acc) = if n=1 then acc else fac1(n-1,n*acc)
```



Rekursiver Aufruf = Rückgabewert
$\Longrightarrow$ Keine Berechnung nach dem Aufruf
$\Longrightarrow$ Lokale Variablen werden nicht mehr gebraucht
$\Longrightarrow$ müssen nicht gespeichert werden

$\Longrightarrow$ Tail-Rekursion hat das beste Speicherverhalten. Es wird zur wilden Rekursion hin immer schlechter.

Eliminierung der repetitiven Rekursion

- Typische Erscheinung **repetitiver Rekursion**:

I.A.: `fun f x = if x = <`x_0`> then <`v_0`> else <e(x,f(g(x)))>`

Bsp.: `fun fac x = if x = 1 then 1 else x*(fac(x-1))`

$\Longrightarrow$ x_0=1, v_0=1, g(x)=x-1 und e(x,y)=x*y

Eliminierung der repetitiven Rekursion

- Typische Erscheinung **repetitiver Rekursion**:

I.A.: `fun f x = if x = <`x_0`> then <`v_0`> else <e(x,f(g(x)))>`

Bsp.: `fun fac x = if x = 1 then 1 else x*(fac(x-1))`

$\Longrightarrow$ x_0=1, v_0=1, g(x)=x-1 und e(x,y)=x*y

- **End-rekursive Variante**: mit Zusatz-Param. (Akkumulator)

Bsp.: `fun fac1 (x,a) = if x = 1 then a else fac1(x-1,x*a)`

$\Longrightarrow$ Aufruf mit fac1 (x,1).

I.A.: `fun f1 (x,a) = if x = <`x_0`> then a else f1(g(x),e(x,a))`

$\Longrightarrow$ Aufruf mit f1 (x,v_0).

Eliminierung der repetitiven Rekursion

```
fun f x       = if x = <x0> then <v0> else <e(x,f(g(x)))>
fun f1 (x,a) = if x = <x0> then  a  else f1(g(x),e(x,a))
```

$\Longrightarrow$ Aufruf mit f1 (x, v_0).

$$
\begin{aligned}
f1(x, v_0) &= f1(g(x), e(x, v_0)) = f1(g(g(x)), e(g(x), e(x, v_0))) = \ldots \\
&= e(g^n(x), e(g^{n-1}(x), \ldots e(g(x), e(x, v_0)) \ldots)) \\
&= g^n(x) \,\square_e\, (g^{n-1}(x) \,\square_e \ldots (g(x) \,\square_e\, (x \,\square_e\, v_0)) \ldots)
\end{aligned}
$$

Eliminierung der repetitiven Rekursion

```
fun f x       = if x = <x0> then <v0> else <e(x,f(g(x)))>
fun f1 (x,a) = if x = <x0> then  a  else f1(g(x),e(x,a))
```

$\Longrightarrow$ Aufruf mit f1 (x, v_0).

$$
\begin{aligned}
f1(x, v_0) &= f1(g(x), e(x, v_0)) = f1(g(g(x)), e(g(x), e(x, v_0))) = \ldots \\
&= e(g^n(x), e(g^{n-1}(x), \ldots e(g(x), e(x, v_0)) \ldots)) \\
&= g^n(x) \,\square_e\, (g^{n-1}(x) \,\square_e \ldots (g(x) \,\square_e\, (x \,\square_e\, v_0)) \ldots)
\end{aligned}
$$

$$
\begin{aligned}
f(x) &= e(x, e(g(x), e(g(g(x), \ldots e(g^n(x), v_0) \ldots))) \\
&= x \,\square_e\, (g(x) \,\square_e\, (g^2(x) \,\square_e \ldots (g^n(x) \,\square_e\, v_0) \ldots))
\end{aligned}
$$

Eliminierung der repetitiven Rekursion

```
fun f x       = if x = <x0> then <v0> else <e(x,f(g(x)))>
fun f1 (x,a) = if x = <x0> then  a  else f1(g(x),e(x,a))
```

$\Longrightarrow$ Aufruf mit f1 (x, v_0).

$$
\begin{aligned}
f1(x, v_0) &= f1(g(x), e(x, v_0)) = f1(g(g(x)), e(g(x), e(x, v_0))) = \ldots \\
&= e(g^n(x), e(g^{n-1}(x), \ldots e(g(x), e(x, v_0)) \ldots)) \\
&= g^n(x) \,\square_e\, (g^{n-1}(x) \,\square_e \ldots (g(x) \,\square_e\, (x \,\square_e\, v_0)) \ldots)
\end{aligned}
$$

$$
\begin{aligned}
f(x) &= e(x, e(g(x), e(g(g(x), \ldots e(g^n(x), v_0) \ldots))) \\
&= x \,\square_e\, (g(x) \,\square_e\, (g^2(x) \,\square_e \ldots (g^n(x) \,\square_e\, v_0) \ldots))
\end{aligned}
$$

$\Longrightarrow$ $f(x) = f1(x, v_0)$, wenn $\square_e$ = assoziativ, kommutativ, z.B.:

```
n*((n-1)*(...*(2*1))) = 1*(2*(...*((n-1)*n)))
```

Rekursionsarten

- Vorsicht bei Listenfunktionen:

```
fun f  x      = if x=0 then nil  else x::f(x-1)
fun f1 (x,a) = if x=0 then a    else f1(x-1,x::a)
```

- Der Listenkonstruktor `::` ist weder kommutativ noch assoziativ.
  $\Longrightarrow$`f x` und `f1 (x,nil)` berechnen nicht den gleichen Wert:

```
- f 5;
val it = [5,4,3,2,1] : int list


- f1(5,nil);
val it = [1,2,3,4,5] : int list
```

Rekursionsarten

- Selbst baumartige Rekursion kann manchmal linearisiert werden:

```
fun fib n = case n of 0 => 0
                    | 1 => 1
                    | n => fib (n-1) + fib (n-2)

fun fib1 n =
    let fun iter (m,f1,f2) =
            if m = n then f1 else iter(m+1,f2,f1+f2)
    in iter(0,0,1)
    end
```

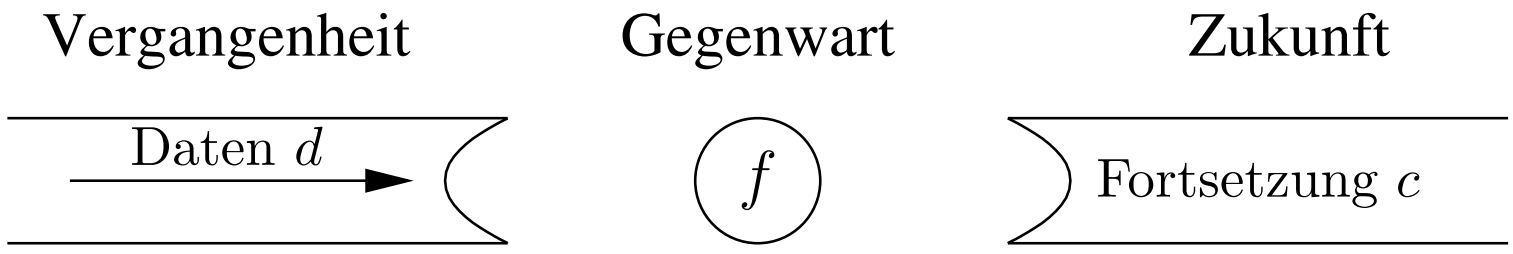
- Das läßt sich aber nicht so einfach verallgemeinern.

Continuation-passing style

Continuation-passing style (CPS)

(≈ *Fortsetzungen-Durchreichen-Programmierstil*):

Abstraktion einer Berechnung:

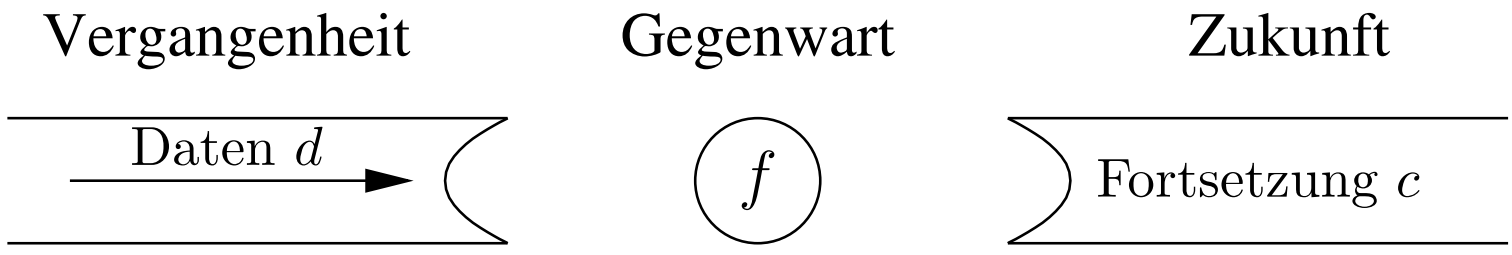


Continuation-passing style

Continuation-passing style (CPS)

(≈ *Fortsetzungen-Durchreichen-Programmierstil*):

Abstraktion einer Berechnung:



Vergangenheit = Daten d, die der aktuellen Bearbeitungsfunktion übergeben werden

Gegenwart = Eine *aktuelle* Bearbeitungsfunktion f

Zukunft = Eine *Fortsetzungsfunktion* c, die den Rest der Berechnung beschreibt (**continuation**)

Die Berechnung liefert $c(f(d))$.

Continuation-passing Style

- Eine Funktion f im CPS:
  - statt einen Wert v zurückzuliefern...
  - .. ruft eine Continuation k mit v auf (k = explizite Angabe, wie die Berechnung fortgesetzt werden soll);

  $\Longrightarrow$ f bekommt die Continuation(s) als zusätzliche Parameter.

Continuation-passing Style

- Eine Funktion f im CPS:
  - statt einen Wert v zurückzuliefern...
  - .. ruft eine Continuation k mit v auf (k = explizite Angabe, wie die Berechnung fortgesetzt werden soll);

  $\Longrightarrow$ f bekommt die Continuation(s) als zusätzliche Parameter.

- Berechnung von $a + b * c$ **ohne Continuations**:

```
fun add (x,y) = x + y
fun mult (x,y) = x * y
fun compute (a,b,c) = add(a,mult(b,c))
```

Continuation-passing Style

- Eine Funktion f im CPS:
  - statt einen Wert v zurückzuliefern...
  - .. ruft eine Continuation k mit v auf (k = explizite Angabe, wie die Berechnung fortgesetzt werden soll);

  $\Longrightarrow$ f bekommt die Continuation(s) als zusätzliche Parameter.

- Berechnung von $a + b * c$ **ohne Continuations**:

```
fun add (x,y) = x + y
fun mult (x,y) = x * y
fun compute (a,b,c) = add(a,mult(b,c))
```

- Berechnung von $a + b * c$ **im CPS**:

```
fun add1 ((x,y),k) = k (add (x,y))
fun mult1((x,y),k) = k (mult (x,y))
fun compute ((a,b,c),k) =
                  mult1((b,c),fn x => add1((a,x),k))
```

Continuation Passing Style: Beispiel

- Berechnung von $a * b + c * d$ mit CPS:

```
fun add1 ((x,y),k) = k (add (x,y))
val add1 = fn : (int * int) * (int -> 'a) -> 'a
fun mult1((x,y),k) = k (mult (x,y))
val mult1 = fn : (int * int) * (int -> 'a) -> 'a

fun compute ((a,b,c,d),k) =
   mult1((a,b),
         fn x => mult1((c,d),
                       fn y => add1((x,y),k)))
val compute = fn : (int * int * int * int) * (int -> 'a) -> 'a
```

Continuation Passing Style: Beispiel

- Berechnung von $a * b + c * d$ mit CPS:

```
fun add1 ((x,y),k) = k (add (x,y))
val add1 = fn : (int * int) * (int -> 'a) -> 'a
fun mult1((x,y),k) = k (mult (x,y))
val mult1 = fn : (int * int) * (int -> 'a) -> 'a

fun compute ((a,b,c,d),k) =
   mult1((a,b),
         fn x => mult1((c,d),
                       fn y => add1((x,y),k)))
val compute = fn : (int * int * int * int) * (int -> 'a) -> 'a

compute ((1,2,3,4),fn x => x);
val it = 14 : int
```

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 des Kontrollflusses
  - benutzer-definierte Kontrollstrukturen durch explizite Modellierung des Kontrollflusses: z.B. Threads, Korutinen, Ausnahmen

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* = **cc**)

  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

Jede Ausdrucksauswertung eines Programms hat eine **cc**:

- Bsp.: Ausdruck `1 + 2 * 3`

- Implizite Continuations für jeden Teilausdruck:

Ausdruck	Implizite 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

Sei: `fun len l = case l of [] => 0 | h::r => 1 + len r`

Ausdruck	Implizite Continuation
`len [1,2]`	k (abhängig vom Kontext der Auswertung)
`case [1,2] ...`	k
`0`	k
`1 + len [2]`	k
`1`	`fn x => k (x + len [2])`
`len [2]`	`fn x => k (1 + x)`
`[2]`	`fn x => k (1 + len x)`

Implizite Continuations verwenden

Aktuelle **Continuations als first class value** (SML/NJ, Scheme, Python):

- Man kann auf implizite Continuations zugreifen und sie als "first class value" manipulieren.
- Insbesondere kann man **eine Continuation c aktivieren** = Abbruch des normalen Programmablaufs; Fortsetzung mit c.

Implizite Continuations verwenden

Das Modul `SMLofNJ.Cont` 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 Continuation kann mit Hilfe von throw aktiviert werden.

Implizite Continuations verwenden

- `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 verwenden

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

Falls die Auswertung von e:

- **k nicht aktiviert** $\Longrightarrow$
  - callcc (fn k => e) liefert den Wert von e;
  - 1 + der Wert von e wird zurückgeliefert.

Implizite Continuations verwenden

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

Falls die Auswertung von e:

- **k nicht aktiviert** $\Longrightarrow$
  - callcc (fn k => e) liefert den Wert von e;
  - 1 + der Wert von e wird zurückgeliefert.
- **k aktiviert** (via throw k e') $\Longrightarrow$
  - Berechnung wird fortgesetzt als hätte callcc (fn k => e) den Wert von e' zurückgeliefert;
  - 1 + den Wert von e' zurückgeliefert.

Continuations

- Bsp.:

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

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

Continuations und Effizienz: Bsp.

```
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] braucht:

- 5 Multiplikationen: 1*(2*(3*(4*(5*0))));
- 5 rekursive Aufrufe; 5 Rücksprünge.

Continuations und Effizienz: Bsp.

- ```
  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] braucht:
  - 5 Multiplikationen: 1*(2*(3*(4*(5*0))));
  - 5 rekursive Aufrufe; 5 Rücksprünge.

- 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] braucht:
  - keine Multiplikation;
  - 5 rekursive Aufrufe; kein Rücksprung: **liefert direkt** 0.
  ```

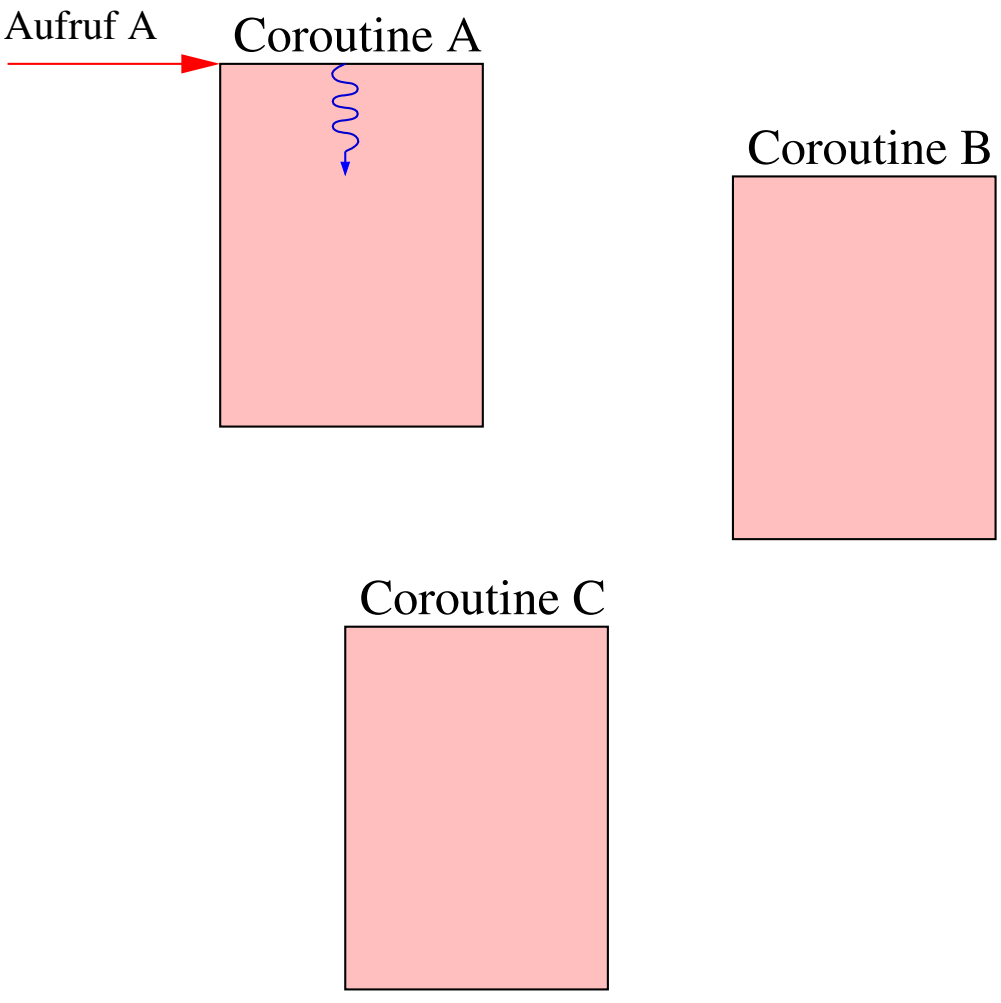
Continuations-Anwendung: Coroutinen

- **Coroutine** = Funktion, die, nach dem sie einen Wert zurückliefern, in dem zuletzt verlassenen Zustand fortgesetzt werden kann.
  - Erster Startpunkt einer Coroutine = Anfangspunkt der Coroutine
  - Nach einer Rückgabe setzt die Berechnung bei einem neuen Aufruf nach dem Rückgabepunkt fort.

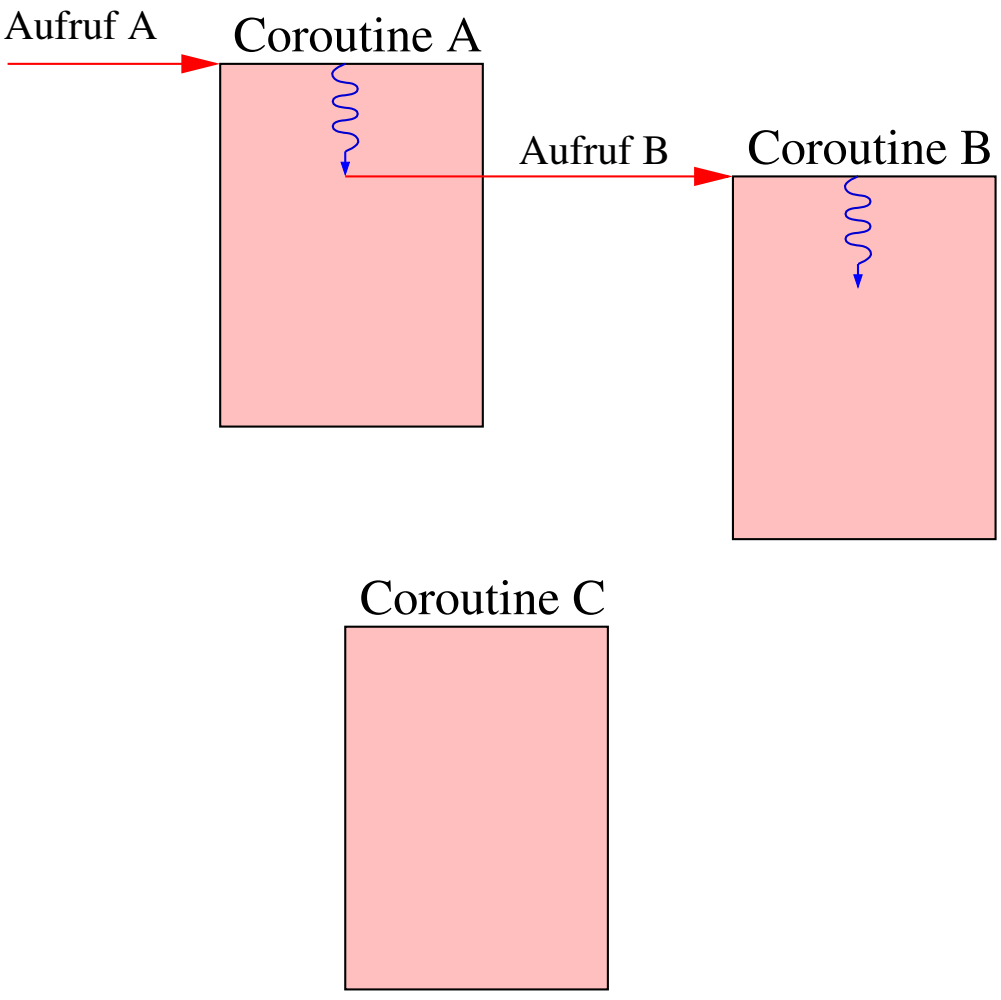
$\Longrightarrow$ Coroutinen = Verallgemeinerung von Funktionen:

  - mehrere Eingangspunkte;
  - mehrere Rückgaben aus einer einzigen Funktionsinstanz.

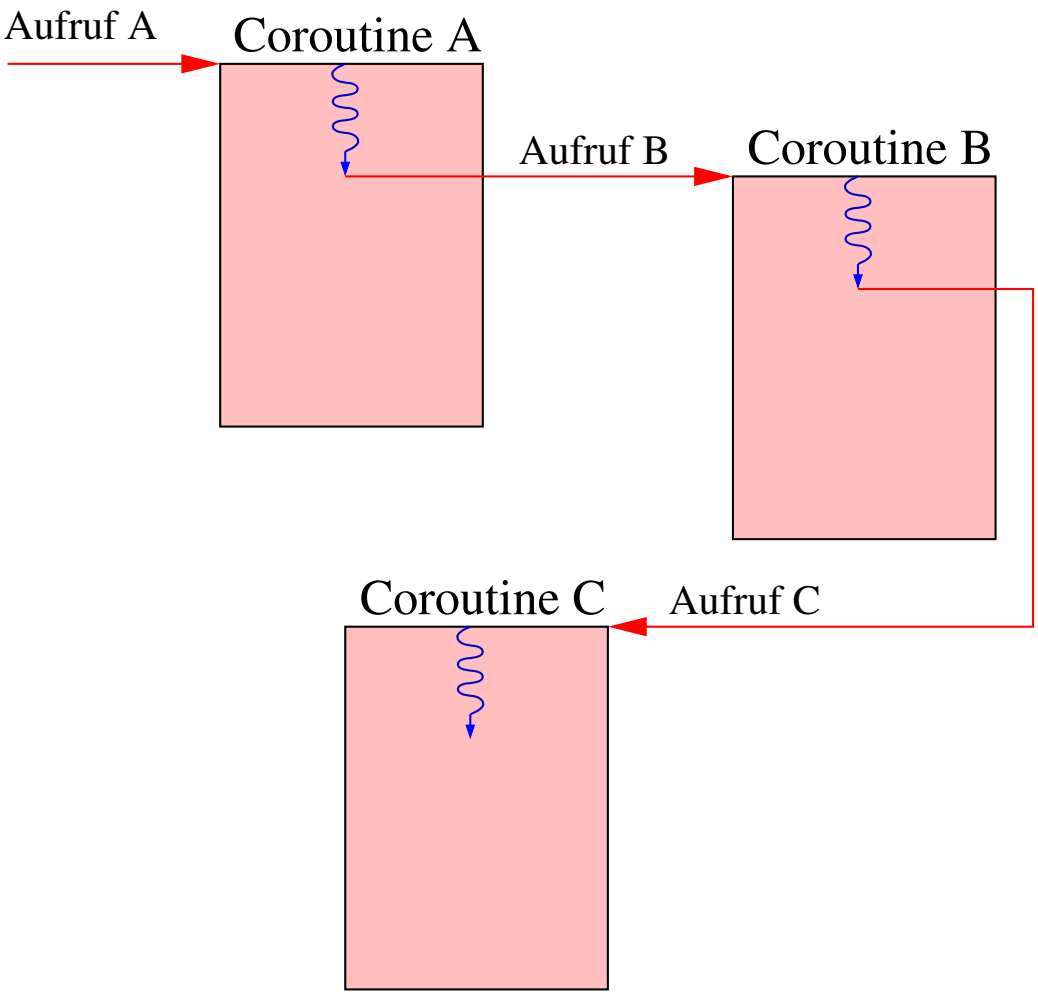
Coroutinen



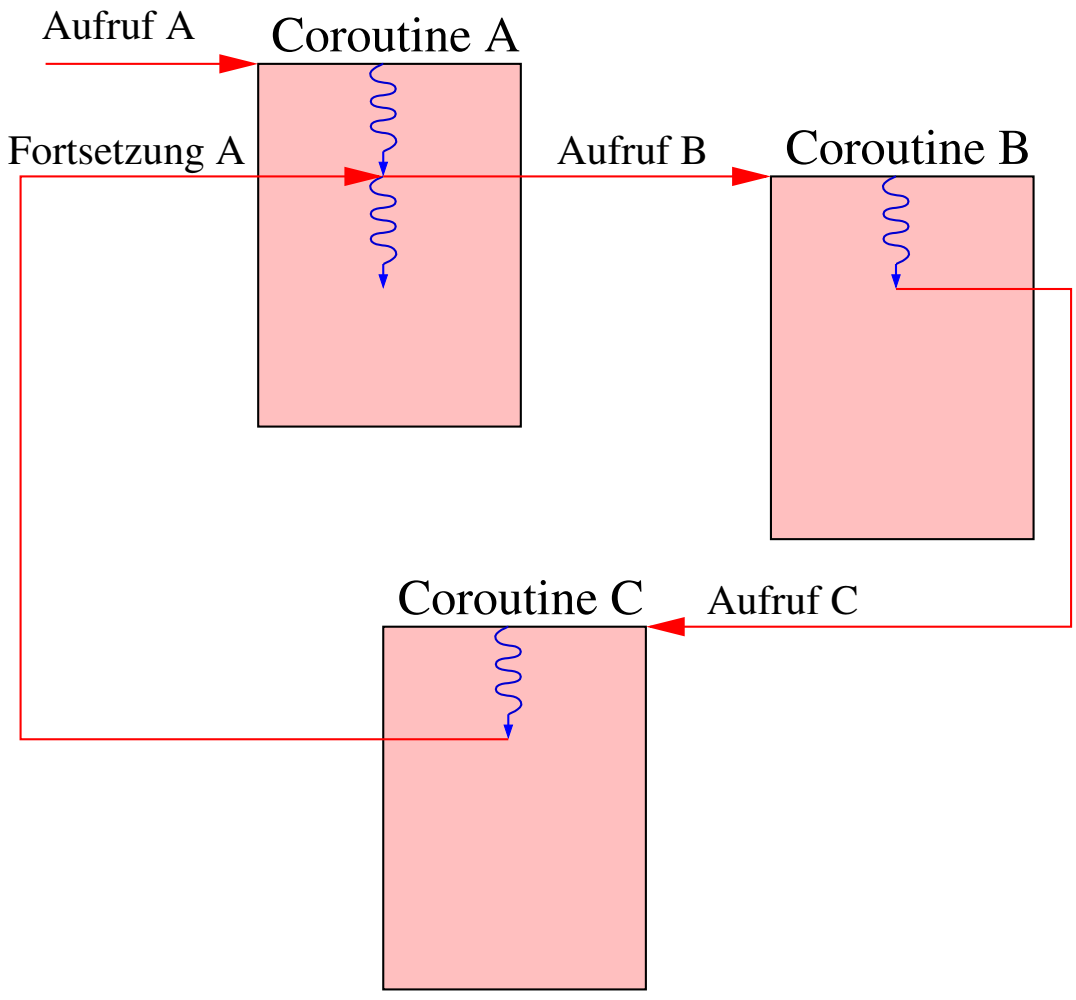
Coroutinen



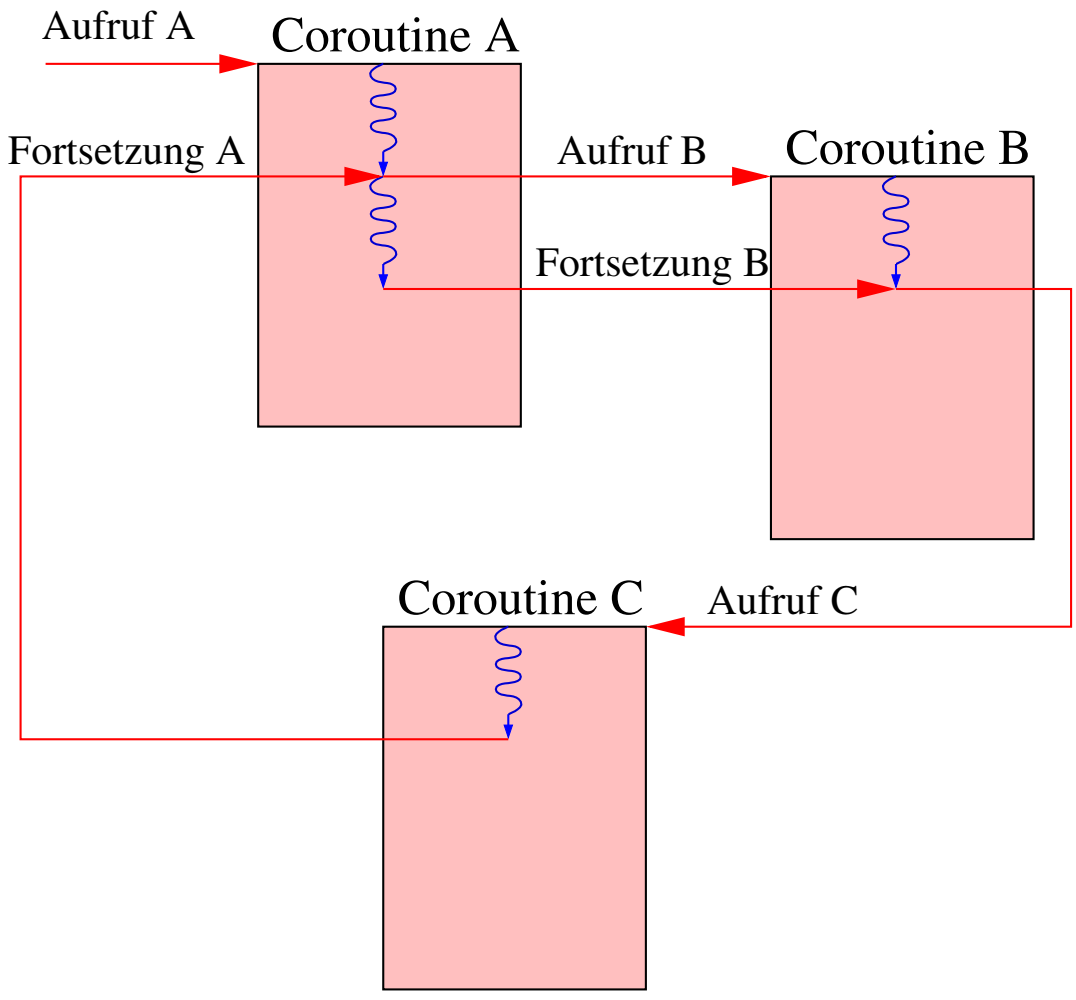
Coroutinen


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

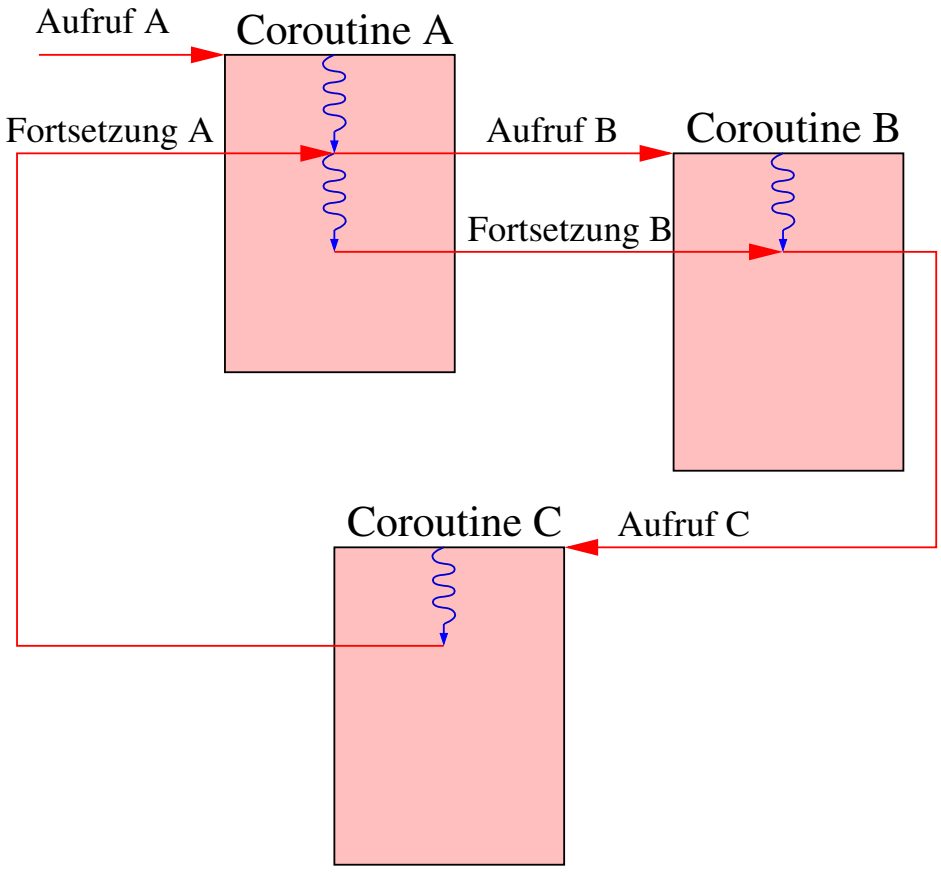
Coroutinen


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**