

Betriebssysteme (BS)

VL 8.3 – Koroutinen und Fäden – Implementierung

Volkmar Sieh / Daniel Lohmann

Lehrstuhl für Informatik 4
Verteilte Systeme und Betriebssysteme

Friedrich-Alexander-Universität
Erlangen Nürnberg

WS 21 – 13. Dezember 2021



https://www4.cs.fau.de/Lehre/WS21/V_BS

Agenda

Motivation

Grundbegriffe

Implementierung

Fortsetzungen

Elementaroperationen

Ausblick

Zusammenfassung

Referenzen



- **Fortsetzung** (engl. *Continuation*): Rest einer Ausführung
 - Eine Fortsetzung ist ein **Objekt**, das einen suspendierten Kontrollfluss repräsentiert.
 - Programmzähler, Register, lokale Variablen, ...
 - kurz: gesamter Kontrollflusszustand
 - wird benötigt, um den Kontrollfluss zu reaktivieren

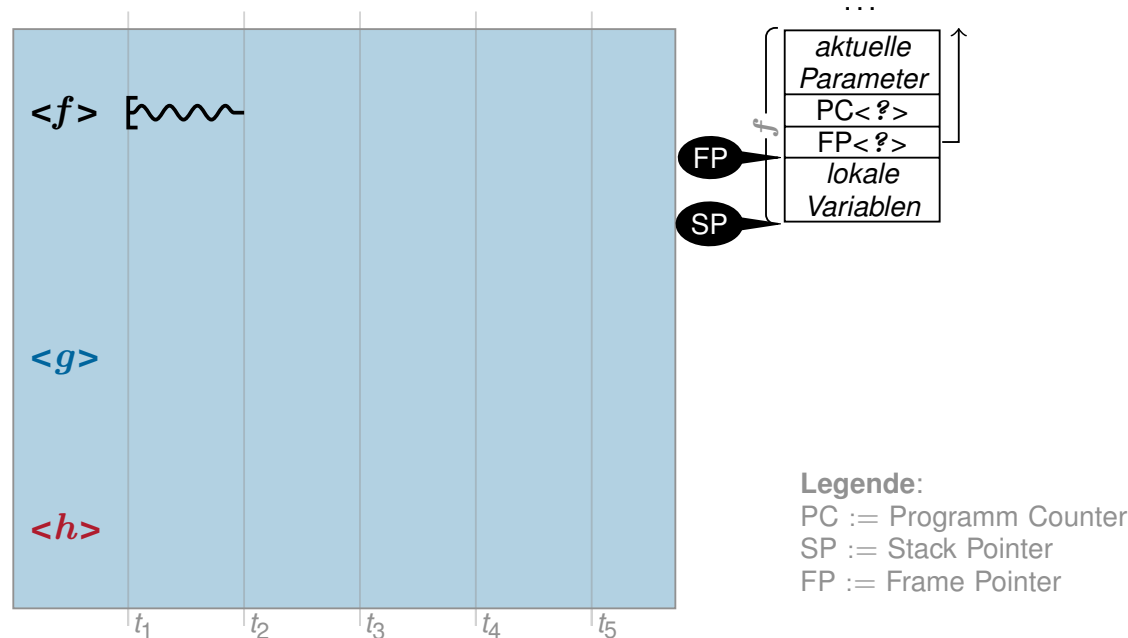
Anmerkung: Fortsetzungen

- Continuations sind ursprünglich entstanden als ein Beschreibungsmittel der **denotationalen Semantik** [3].
- Sprachen wie Haskell oder Scheme bieten Continuations als eigenes Sprachmittel an.



Routinen $\mapsto$ asymmetrisches Fortsetzungsmodell

- Routinen-Fortsetzungen werden i. a. auf einem **Stapel** instantiiert
 - in Form von **Stapel-Rahmen**, erzeugt und zerstört durch
 - **Compiler** (explizit) und **CPU** (implizit) bei *call*, *ret*
 - **Kopplungsfunktion** (explizit) und **CPU** (implizit) bei *↵*, *iret*
 - Der Compiler verwendet dafür i. a. den CPU-Stapel
 - *call*, *ret*, *push*, *pop*, ... verwenden implizit den CPU-Stapel

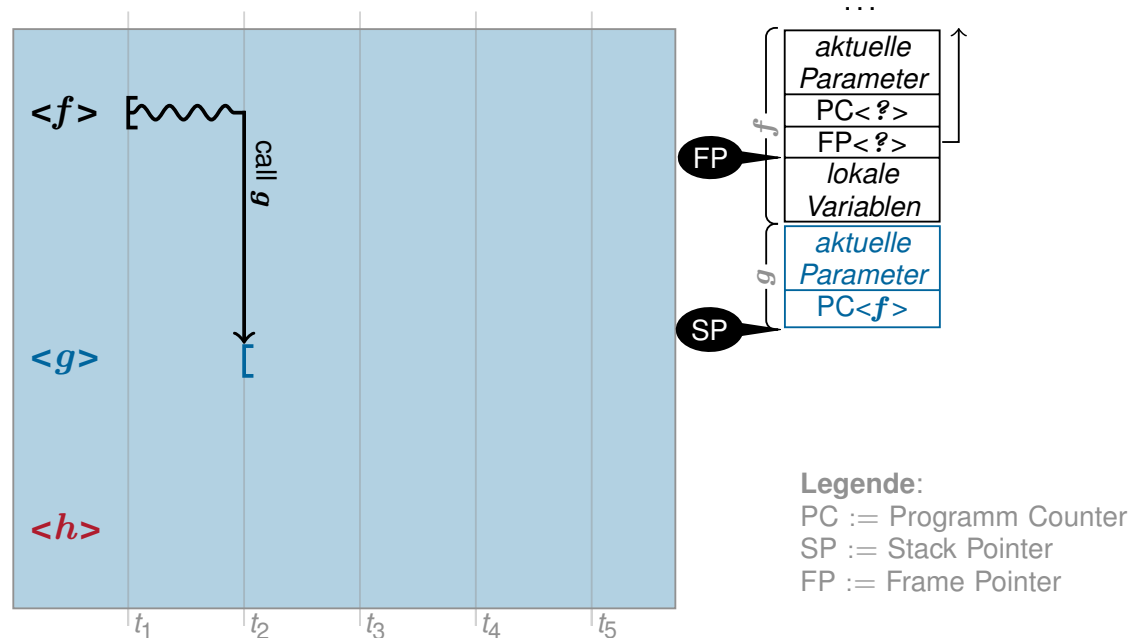


Für jeden Routinen-Kontrollfluss legen **CPU** und **Compiler** einen **Rahmen** an. Dieser enthält die **Fortsetzung** des Aufrufers.



Routinen $\mapsto$ asymmetrisches Fortsetzungsmodell

- Routinen-Fortsetzungen werden i. a. auf einem **Stapel** instantiiert
 - in Form von **Stapel-Rahmen**, erzeugt und zerstört durch
 - **Compiler** (explizit) und **CPU** (implizit) bei *call*, *ret*
 - **Kopplungsfunktion** (explizit) und **CPU** (implizit) bei *↩*, *iret*
 - Der Compiler verwendet dafür i. a. den CPU-Stapel
 - *call*, *ret*, *push*, *pop*, ... verwenden implizit den CPU-Stapel

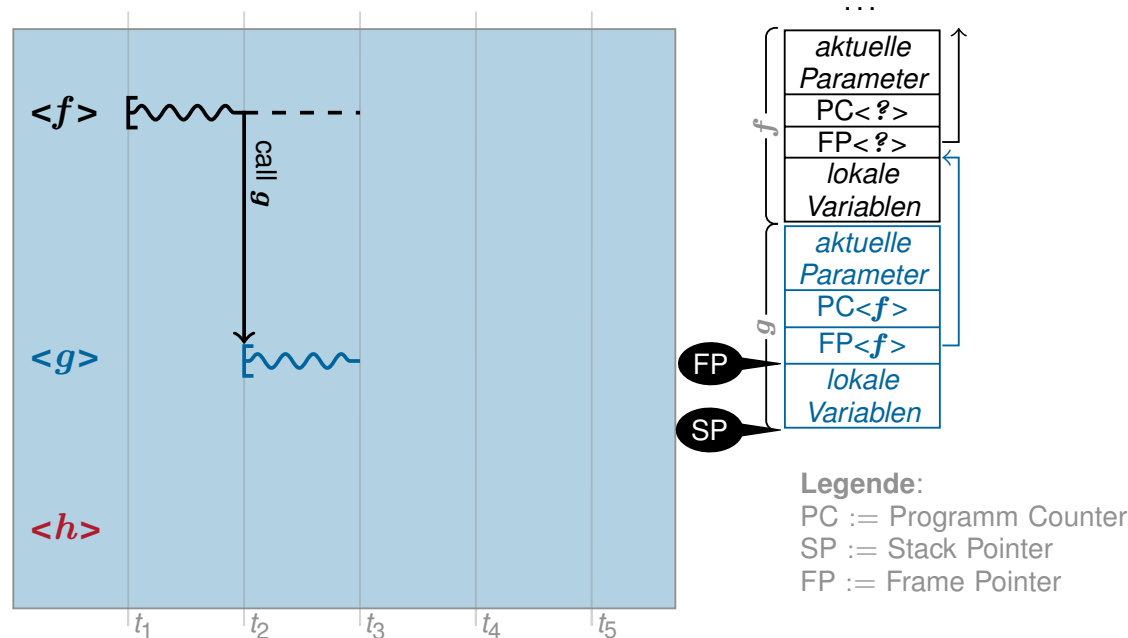


Für jeden Routinen-Kontrollfluss legen **CPU** und **Compiler** einen **Rahmen** an. Dieser enthält die **Fortsetzung** des Aufrufers.



Routinen $\mapsto$ asymmetrisches Fortsetzungsmodell

- Routinen-Fortsetzungen werden i. a. auf einem **Stapel** instantiiert
 - in Form von **Stapel-Rahmen**, erzeugt und zerstört durch
 - **Compiler** (explizit) und **CPU** (implizit) bei *call*, *ret*
 - **Kopplungsfunktion** (explizit) und **CPU** (implizit) bei *↯*, *iret*
 - Der Compiler verwendet dafür i. a. den CPU-Stapel
 - *call*, *ret*, *push*, *pop*, ... verwenden implizit den CPU-Stapel

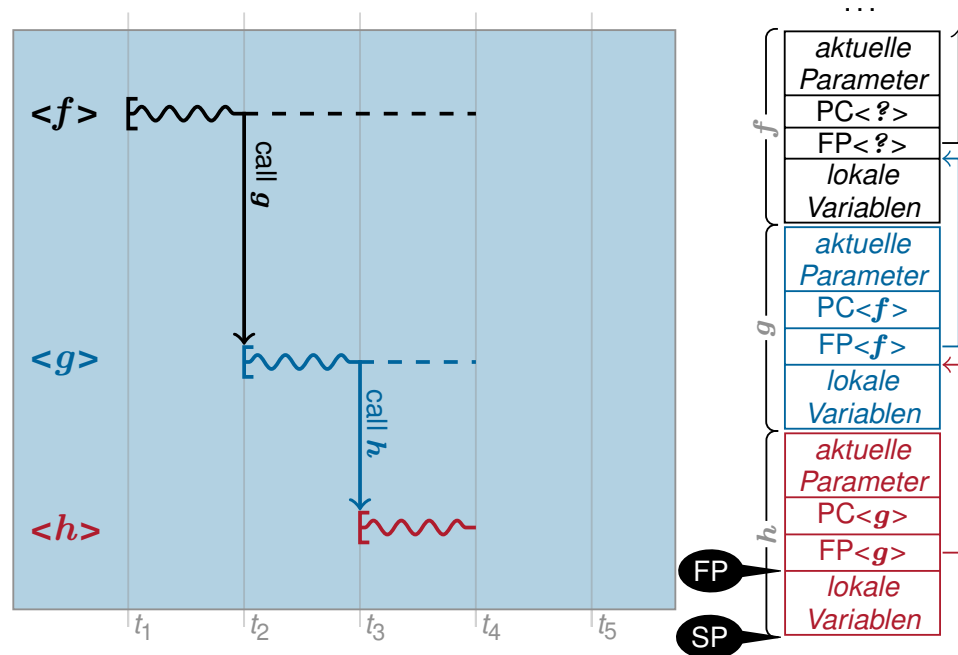


Für jeden Routinen-Kontrollfluss legen **CPU** und **Compiler** einen **Rahmen** an. Dieser enthält die **Fortsetzung** des Aufrufers.



Routinen $\mapsto$ asymmetrisches Fortsetzungsmodell

- Routinen-Fortsetzungen werden i. a. auf einem **Stapel** instantiiert
 - in Form von **Stapel-Rahmen**, erzeugt und zerstört durch
 - **Compiler** (explizit) und **CPU** (implizit) bei *call*, *ret*
 - **Kopplungsfunktion** (explizit) und **CPU** (implizit) bei *↵*, *iret*
 - Der Compiler verwendet dafür i. a. den CPU-Stapel
 - *call*, *ret*, *push*, *pop*, ... verwenden implizit den CPU-Stapel

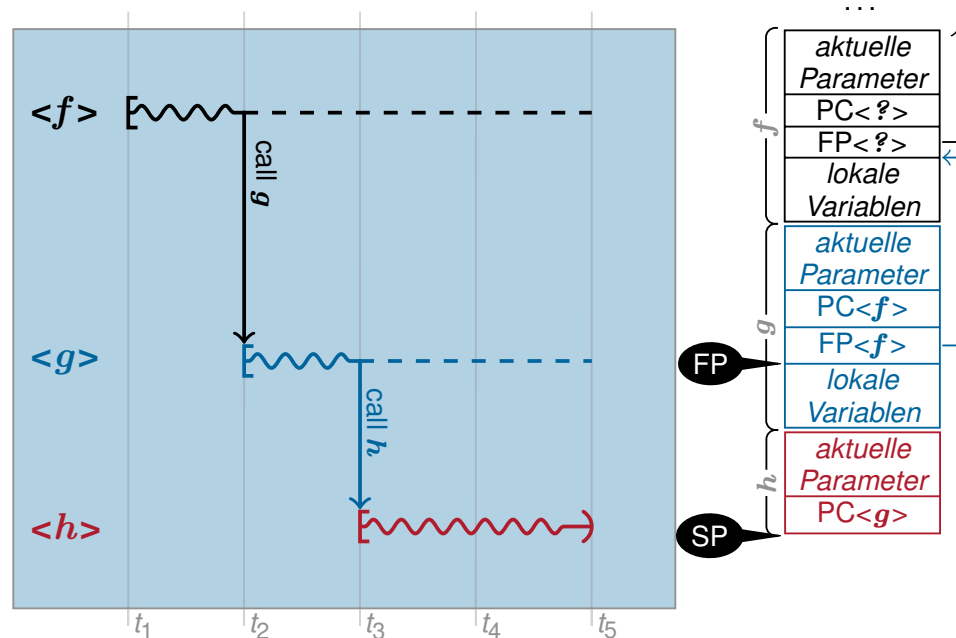


Für jeden Routinen-Kontrollfluss legen **CPU** und **Compiler** einen **Rahmen** an. Dieser enthält die **Fortsetzung** des Aufrufers.



Routinen $\mapsto$ asymmetrisches Fortsetzungsmodell

- Routinen-Fortsetzungen werden i. a. auf einem **Stapel** instantiiert
 - in Form von **Stapel-Rahmen**, erzeugt und zerstört durch
 - **Compiler** (explizit) und **CPU** (implizit) bei *call*, *ret*
 - **Kopplungsfunktion** (explizit) und **CPU** (implizit) bei *↯*, *iret*
 - Der Compiler verwendet dafür i. a. den CPU-Stapel
 - *call*, *ret*, *push*, *pop*, ... verwenden implizit den CPU-Stapel

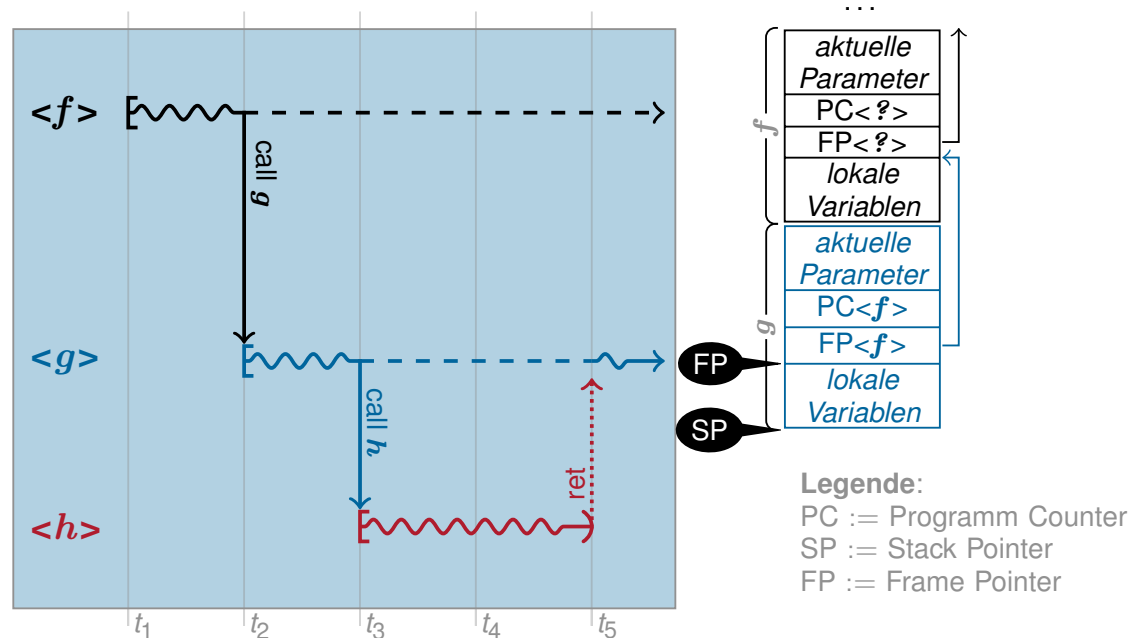


Für jeden Routinen-Kontrollfluss legen **CPU** und **Compiler** einen **Rahmen** an. Dieser enthält die **Fortsetzung** des Aufrufers.



Routinen $\mapsto$ asymmetrisches Fortsetzungsmodell

- Routinen-Fortsetzungen werden i. a. auf einem **Stapel** instantiiert
 - in Form von **Stapel-Rahmen**, erzeugt und zerstört durch
 - **Compiler** (explizit) und **CPU** (implizit) bei *call*, *ret*
 - **Kopplungsfunktion** (explizit) und **CPU** (implizit) bei *↩*, *iret*
 - Der Compiler verwendet dafür i. a. den CPU-Stapel
 - *call*, *ret*, *push*, *pop*, ... verwenden implizit den CPU-Stapel



Für jeden Routinen-Kontrollfluss legen **CPU** und **Compiler** einen **Rahmen** an. Dieser enthält die **Fortsetzung** des Aufrufers.



Koroutinen $\mapsto$ symmetrisches Fortsetzungsmodell

- Koroutinen-Fortsetzungen werden i. a. nicht nativ unterstützt
- **Ansatz:** Koroutinen-Fortsetzungen durch **Routinen-Fortsetzungen** implementieren [2]
 - Ein *resume*-Aufruf sieht für den Compiler wie die Erzeugung und Aktivierung eines ganz normalen Routinen-Kontrollflusses aus.
 - Vor dem `ret` wird in *resume* jedoch intern der Koroutinen-Kontrollfluss gewechselt.
- **Folge:** Technisch gesehen, müssen wir das Routinen-Fortsetzungsmodell **des Compilers** bereitstellen
 - Registerverwendung $\rightsquigarrow$ **nichtflüchte Register** über Wechsel erhalten
 - Fortsetzungs-Stapel $\rightsquigarrow$ **eigener Stapel** für jede Koroutinen-Instanz

Eine Koroutinen-Instanz wird durch ihren Fortsetzungs-Stapel repräsentiert

- während der Ausführung ist dieser Stapel der CPU-Stapel
- oberster Stapel-Rahmen enthält immer die Fortsetzung
- Koroutinen-Wechsel $\mapsto$ Stapel-Wechsel + `ret`



■ Aufgabe: Koroutinen-Kontrollfluss wechseln

```
// Typ fuer Stapelzeiger (Stapel ist Feld von void*)
typedef void**  SP;

extern "C" void resume( SP& from_sp, SP& to_sp ) {
    /* aktueller Stapel-Rahmen ist Fortsetzung des zu
       suspendierenden Kontrollflusses (Aufrufer von resume) */

    < sichere CPU-Stapelzeiger in from_sp >
    < lade CPU-Stapelzeiger aus to_sp >

    /* aktueller Stapel-Rahmen ist Fortsetzung des zu
       reaktivierenden Kontrollflusses */

} // Ruecksprung
```



■ Aufgabe: Koroutinen-Kontrollfluss wechseln

```
// Typ fuer Stapelzeiger (Stapel ist Feld von void*)
typedef void** SP;

extern "C" void resume( SP& from_sp, SP& to_sp ) {
    /* aktueller Stapel-Rahmen ist Fortsetzung des zu
       suspendierenden Kontrollflusses (Aufrufer von resume) */

    < sichere CPU-Stapelzeiger in from_sp >
    < lade CPU-Stapelzeiger aus to_sp >

    /* aktueller Stapel-Rahmen ist Fortsetzung des zu
       reaktivierenden Kontrollflusses */

} // Ruecksprung
```

Problem: nicht-flüchtige Register

- Der Stapel-Rahmen enthält keine **nicht-flüchtigen Register**, da der Aufrufer davon ausgeht, dass diese nicht verändert werden.
- Wir springen jedoch in einen **anderen Aufrufer** zurück!



Implementierung: *resume*

- **Problem:** nicht-flüchtige Register
 - Routinen-Fortsetzung enthält keine nicht-flüchtigen Register
 - $\rightsquigarrow$ diese müssen explizit **gesichert** und **restauriert** werden
- Viele Implementierungsvarianten sind denkbar
 - nicht-flüchtige Register in eigener Struktur sichern ($\rightsquigarrow$ Übung)
 - oder einfach als „lokale Variablen“ auf dem Stapel:

```
extern "C" void resume( SP& from_sp, SP& to_sp ) {  
    /* aktueller Stapel-Rahmen ist Fortsetzung des zu  
       suspendierenden Kontrollflusses (Aufrufer von resume) */  
    <lege nicht-fluechtige Register auf den Stapel >  
    < sichere CPU-Stapelzeiger in from_sp >  
    < lade CPU-Stapelzeiger aus to_sp >  
    <hole nicht-fluechtige Register vom Stapel >  
    /* aktueller Stapel-Rahmen ist Fortsetzung des zu  
       reaktivierenden Kontrollflusses */  
}  
// Ruecksprung
```



Implementierung: *resume*

- Implementierung vom *resume* ist architekturabhängig
 - Aufbau der Stapel-Rahmen
 - nicht-flüchtige Register
 - Wachstumsrichtung des Stapels
- Außerdem muss man Register bearbeiten $\leadsto$ **Assembler**

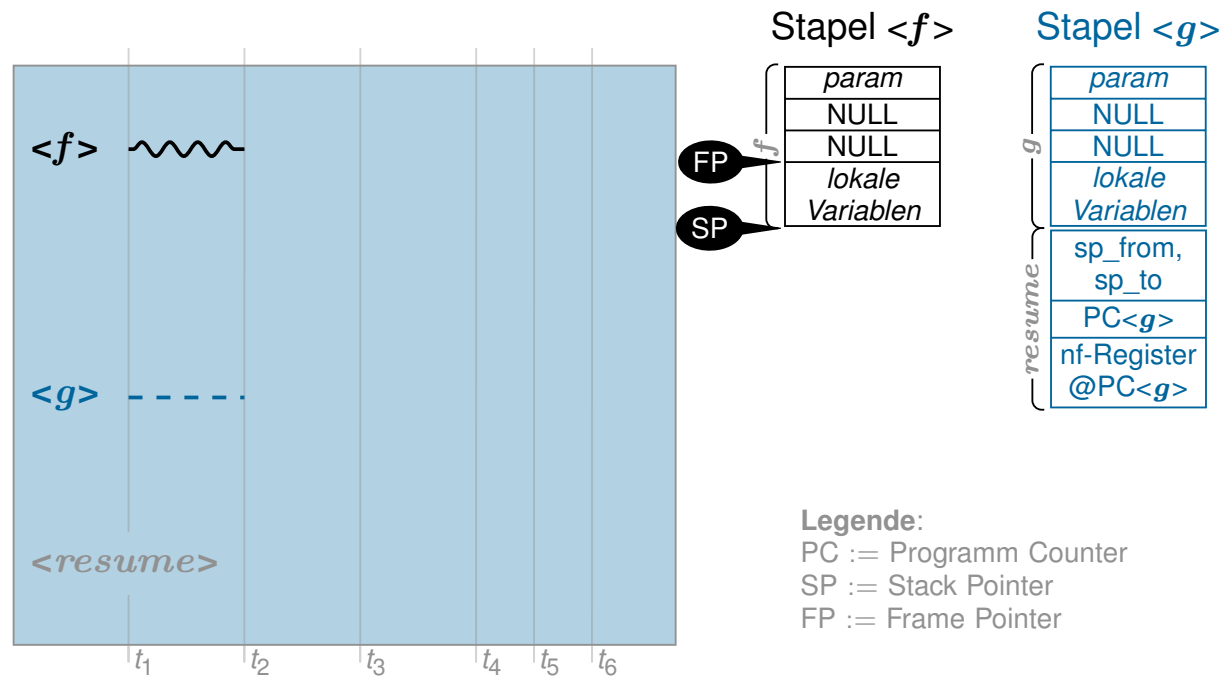
Beispiel Motorola 68000:

```
// extern "C" void resume( SP& sp_from, SP& sp_to )
resume:
    move.l    4(sp), a0           // a0 = &sp_from
    move.l    8(sp), a1           // a1 = &sp_to
    movem.l   d2-d7/a2-a6, -(sp)  // nf-Register auf den Stapel
    move.l    sp, (a0)            // sp_from = sp
    move.l    (a1), sp            // sp = sp_to
    movem.l   (sp)+, d2-d7/a2-a6  // hole nf-Register vom Stapel
    rts                          // "Ruecksprung"
```



Beispiel: Verwendung von *resume*

Koroutinen-Kontrollfluss $\langle f \rangle$ übergibt an $\langle g \rangle$:

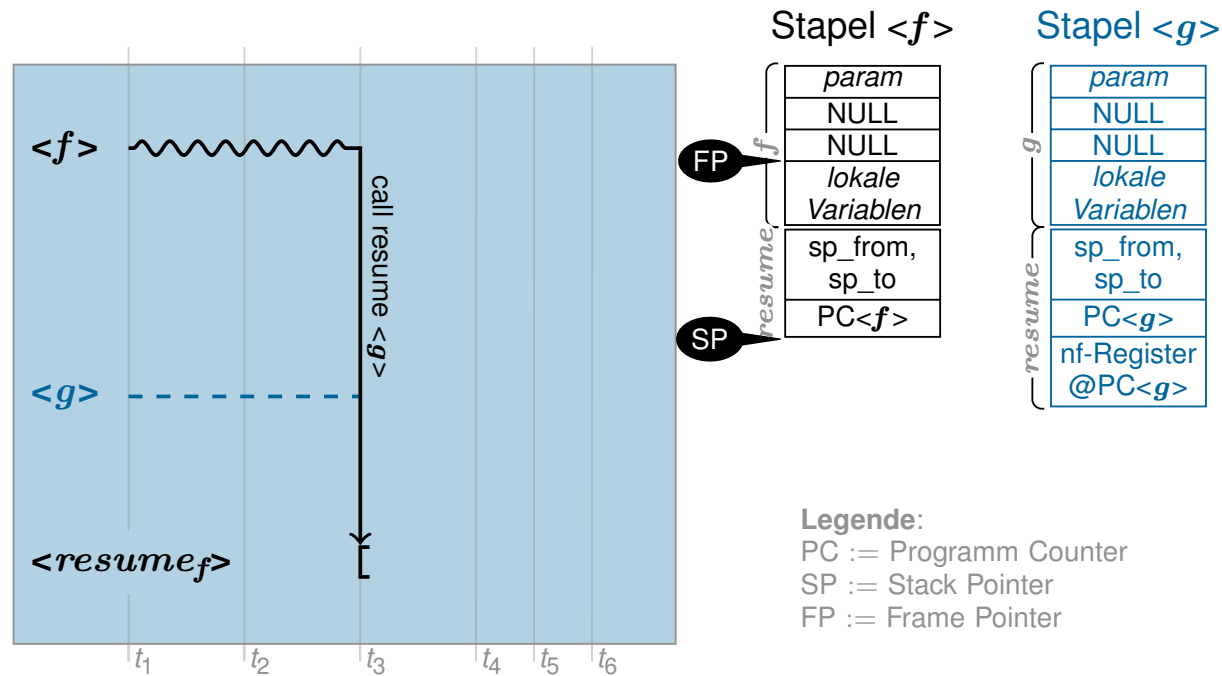


1. Koroutine $\langle f \rangle$ ist aktiv, Koroutine $\langle g \rangle$ ist suspendiert



Beispiel: Verwendung von *resume*

Koroutinen-Kontrollfluss $\langle f \rangle$ übergibt an $\langle g \rangle$:

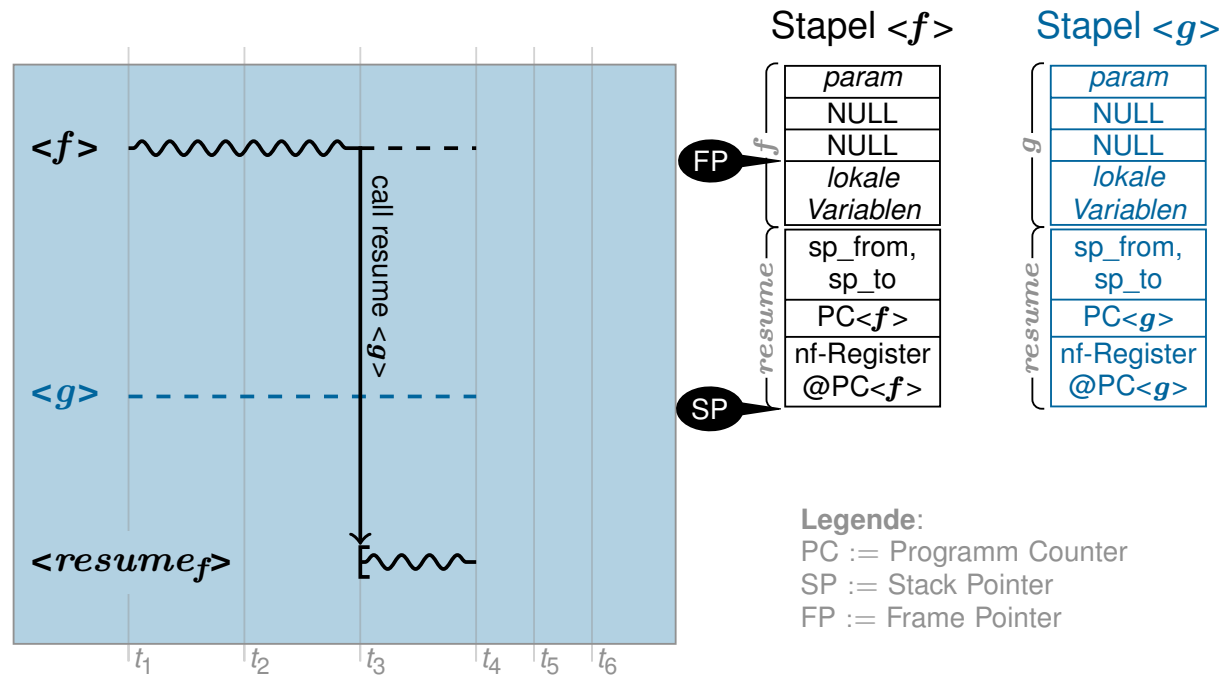


1. Koroutine $\langle f \rangle$ ist aktiv, Koroutine $\langle g \rangle$ ist suspendiert
2. $\langle f \rangle$ instantiiert den Routinen-Kontrollfluss $\langle resume_f \rangle$ und legt dazu Parameter (Stapelvariablen von $\langle f \rangle$ und $\langle g \rangle$) sowie die Rücksprung-Adresse ($\hookrightarrow$ Fortsetzung von $\langle f \rangle$) auf den Stapel.



Beispiel: Verwendung von *resume*

Koroutinen-Kontrollfluss $\langle f \rangle$ übergibt an $\langle g \rangle$:

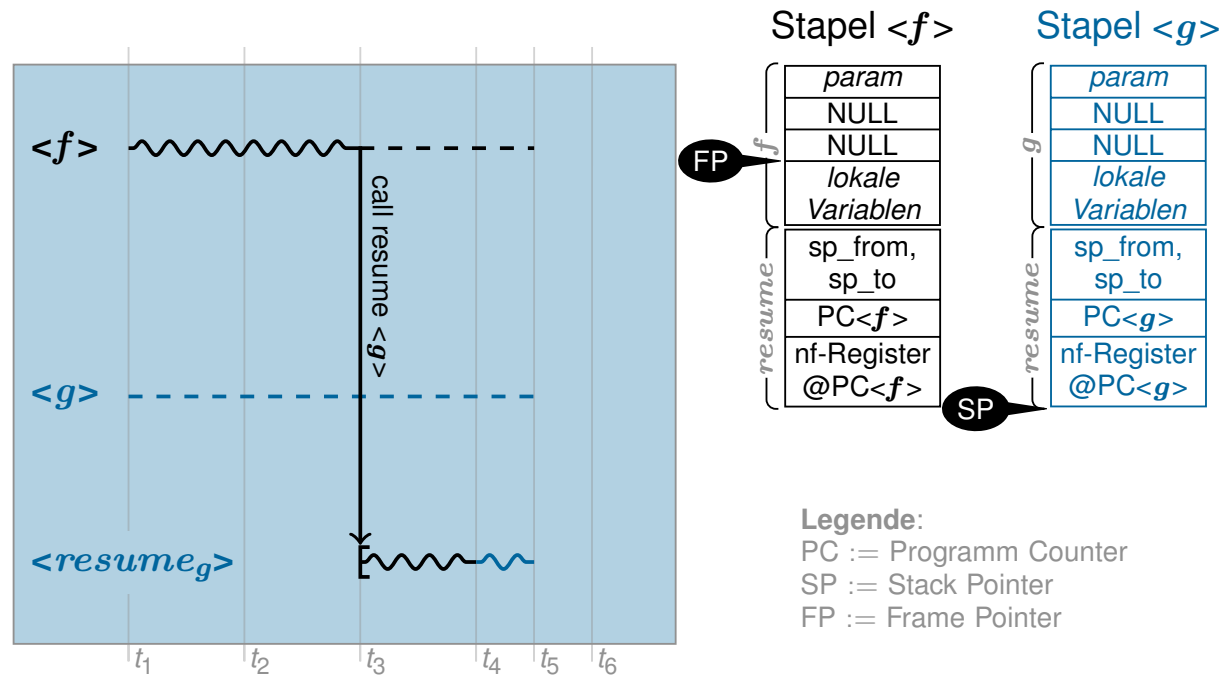


1. Koroutine $\langle f \rangle$ ist aktiv, Koroutine $\langle g \rangle$ ist suspendiert
2. $\langle f \rangle$ instantiiert den Routinen-Kontrollfluss *resume_f* und legt dazu Parameter (Stapelvariablen von $\langle f \rangle$ und $\langle g \rangle$) sowie die Rücksprung-Adresse ($\hookrightarrow$ Fortsetzung von $\langle f \rangle$) auf den Stapel.
3. *resume_f* sichert nicht-flüchtige Register von $\langle f \rangle$ auf dem Stapel und eigenen SP in *sp_from*



Beispiel: Verwendung von *resume*

Koroutinen-Kontrollfluss $\langle f \rangle$ übergibt an $\langle g \rangle$:

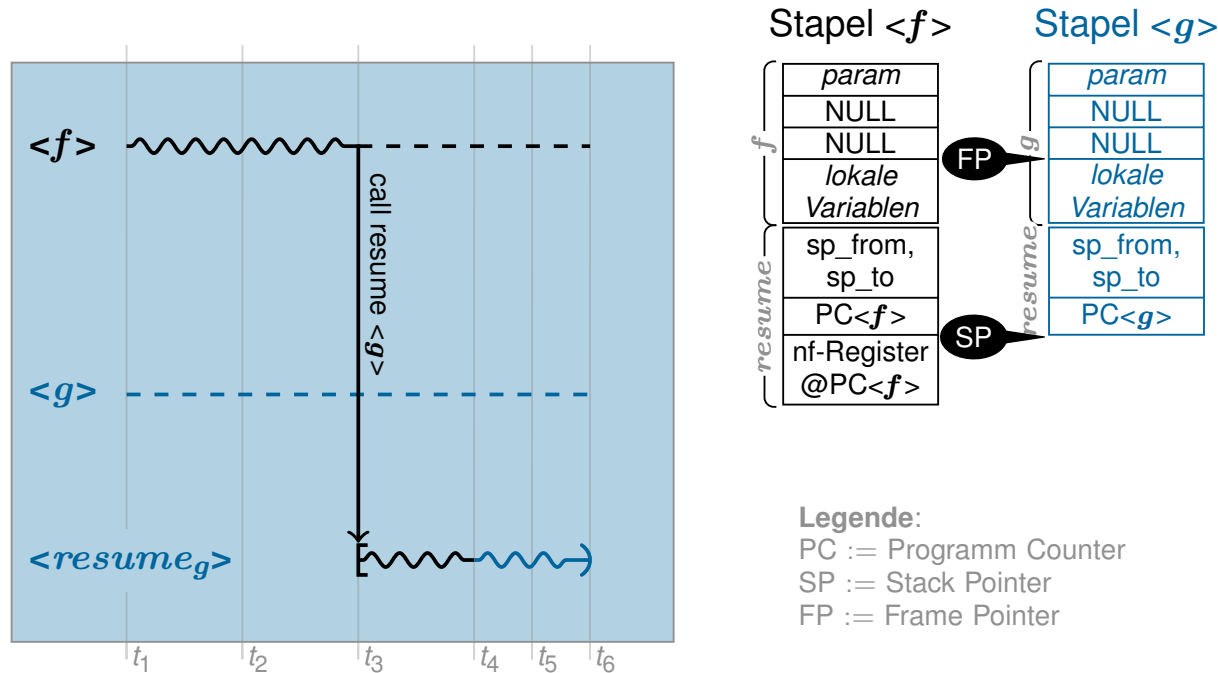


1. Koroutine $\langle f \rangle$ ist aktiv, Koroutine $\langle g \rangle$ ist suspendiert
2. $\langle f \rangle$ instantiiert den Routinen-Kontrollfluss $\langle \text{resume}_f \rangle$ und legt dazu Parameter (Stapelvariablen von $\langle f \rangle$ und $\langle g \rangle$) sowie die Rücksprung-Adresse ($\mapsto$ Fortsetzung von $\langle f \rangle$) auf den Stapel.
3. $\langle \text{resume}_f \rangle$ sichert nicht-flüchtige Register von $\langle f \rangle$ auf dem Stapel und eigenen SP in *sp_from*
4. Wechsel des SP auf den Stapel von $\langle g \rangle$ (*sp_to*) $\rightsquigarrow$ **Koroutinen-Wechsel**, nun läuft $\langle \text{resume}_g \rangle$



Beispiel: Verwendung von *resume*

Koroutinen-Kontrollfluss $\langle f \rangle$ übergibt an $\langle g \rangle$:

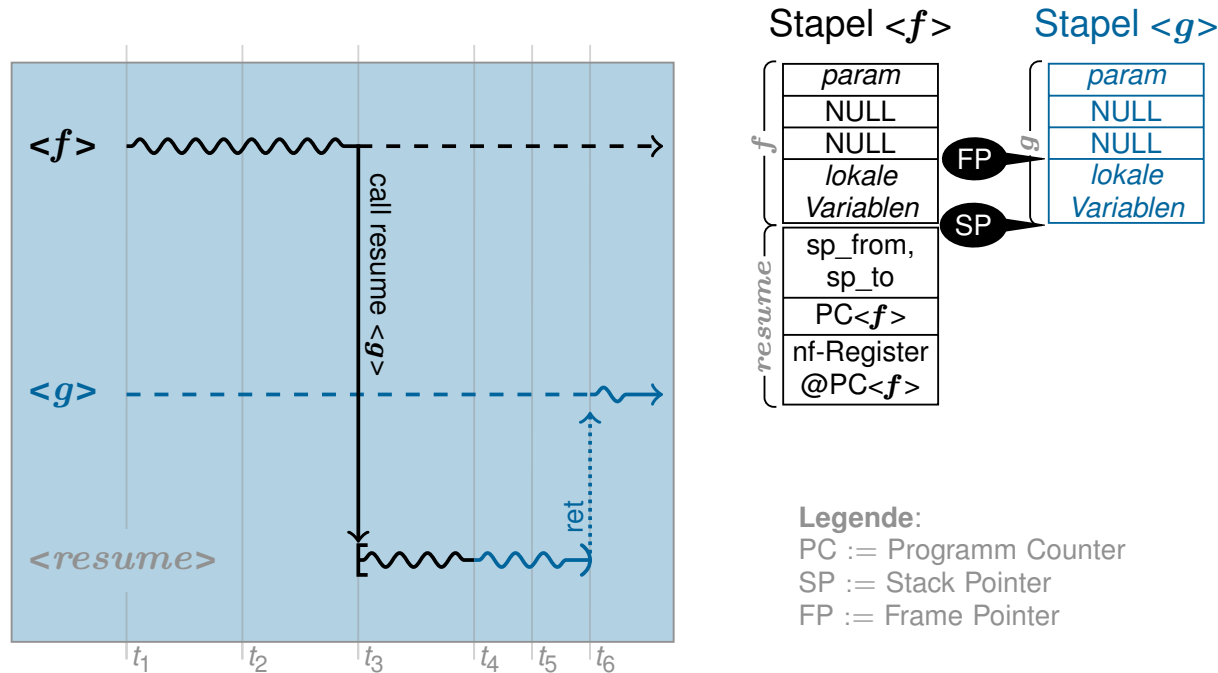


1. Koroutine $\langle f \rangle$ ist aktiv, Koroutine $\langle g \rangle$ ist suspendiert
2. $\langle f \rangle$ instantiiert den Routinen-Kontrollfluss $\langle resume_f \rangle$ und legt dazu Parameter (Stapelvariablen von $\langle f \rangle$ und $\langle g \rangle$) sowie die Rücksprung-Adresse ($\rightarrow$ Fortsetzung von $\langle f \rangle$) auf den Stapel.
3. $\langle resume_f \rangle$ sichert nicht-flüchtige Register von $\langle f \rangle$ auf dem Stapel und eigenen SP in sp_from
4. Wechsel des SP auf den Stapel von $\langle g \rangle$ (sp_to) $\rightsquigarrow$ **Koroutinen-Wechsel**, nun läuft $\langle resume_g \rangle$
5. $\langle resume_g \rangle$ holt nicht-flüchtige Register von $\langle g \rangle$ vom Stapel.



Beispiel: Verwendung von *resume*

Koroutinen-Kontrollfluss $\langle f \rangle$ übergibt an $\langle g \rangle$:



1. Koroutine $\langle f \rangle$ ist aktiv, Koroutine $\langle g \rangle$ ist suspendiert
2. $\langle f \rangle$ instantiiert den Routinen-Kontrollfluss $\langle resume_f \rangle$ und legt dazu Parameter (Stapelvariablen von $\langle f \rangle$ und $\langle g \rangle$) sowie die Rücksprung-Adresse ($\rightarrow$ Fortsetzung von $\langle f \rangle$) auf den Stapel.
3. $\langle resume_f \rangle$ sichert nicht-flüchtige Register von $\langle f \rangle$ auf dem Stapel und eigenen SP in sp_from
4. Wechsel des SP auf den Stapel von $\langle g \rangle$ (sp_to) $\rightsquigarrow$ **Koroutinen-Wechsel**, nun läuft $\langle resume_g \rangle$
5. $\langle resume_g \rangle$ holt nicht-flüchtige Register von $\langle g \rangle$ vom Stapel.
6. Routinen-Kontrollfluss $\langle resume_g \rangle$ terminiert mit ret: $\langle g \rangle$ ist aktiv, $\langle f \rangle$ ist suspendiert



■ Aufgabe: Koroutinen-Kontrollfluss *<start>* erzeugen

■ Gebraucht wird dafür

1. **Stapelspeicher** (irgendwo, global) `static void* stack_start[ 256 ];`
2. **Stapelzeiger** `SP sp_start = &stack_start[ 256 ];`
3. **Startfunktion** `void start( void* param ) ...`
4. **Parameter** für die Startfunktion

■ Koroutinen-Kontrollfluss wird suspendiert erzeugt

■ Ansatz: *create* erzeugt zwei Stapel-Rahmen

■ so als hätte *<start>* bereits *resume* als **Routine** aufgerufen

1. Rahmen der Startfunktion selber (erzeugt vom „virtuellen Aufrufer“)
2. Rahmen von *resume* (enthält Fortsetzung in *<start>*)

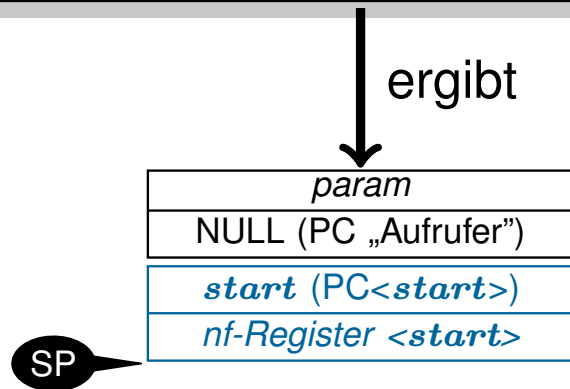
■ erstes *resume* macht „Rücksprung“ an den Beginn von *start*



Implementierung: *create*

Beispiel Motorola 68000:

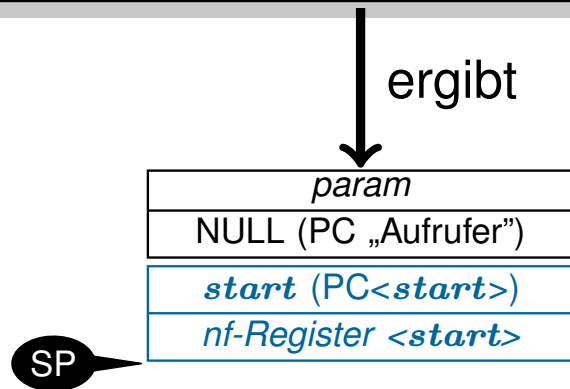
```
void create( SP& sp_new, void (*start)(void*), void* param) {  
    *(--sp_new) = param;    // Parameter von Startfunktion  
    *(--sp_new) = 0;        // Aufrufer (gibt es nicht!)  
  
    *(--sp_new) = start;    // Startadresse  
    sp_new -= 11;           // nicht-fluechtige Register (Werte egal)  
}
```



Implementierung: *create*

Beispiel Motorola 68000:

```
void create( SP& sp_new, void (*start)(void*), void* param) {  
    * (--sp_new) = param; // Parameter von Startfunktion  
    * (--sp_new) = 0;     // Aufrufer (gibt es nicht!)  
  
    * (--sp_new) = start; // Startadresse  
    sp_new -= 11;         // nicht-fluechtige Register (Werte egal)  
}
```



Da der Rücksprung an den **Anfang** einer Funktion erfolgt, sind die Rahmen sehr einfach aufgebaut.

Zu diesem Fortsetzungspunkt hat ein Routinen-Kontrollfluss noch:

- keinen FP verwendet oder gesichert
- keine lokalen Variablen auf dem Stapel angelegt
- keine Annahmen über den Inhalt von nf-Registern



Implementierung: *destroy*

- **Aufgabe:** Koroutinen-Kontrollfluss zerstören
- **Ansatz:** Kontrollfluss-Kontext freigeben
 - entspricht Freigabe der Kontextvariablen ($\mapsto$ Stapelzeiger)
 - Stapelspeicher kann anschließend anderweitig verwendet werden

Das ist wenigstens mal einfach :-)

