(3) Differenzen für Mengen

Betrachte die Fixpunkt-Berechnung:

$$
\begin{array}{l}
x = \emptyset; \\
\textsf{for} \;\; (t = F\,x; t \not\subseteq x; \boxed{t = F\,x;}) \\
\qquad x = x \cup t;
\end{array}
$$

Ist F distributiv, könnte man sie ersetzen durch:

$$
\begin{array}{l}
x = \emptyset; \\
\textsf{for} \;\; (\Delta = F\,x; \Delta \neq \emptyset; \boxed{\Delta = (F\,\Delta) \setminus x;}) \\
\qquad x = x \cup \Delta;
\end{array}
$$

Die Funktion F muss jetzt nur noch für die kleineren Mengen Δ ausgerechnet werden :-) semi-naive Iteration

$$
\begin{array}{lcl}
\text{Statt der Folge:} & & \emptyset \;\subseteq\; F\,(\emptyset) \;\subseteq\; F^2\,(\emptyset) \;\subseteq\; \ldots \\
\text{berechnen wir:} & & \Delta_1 \;\cup\; \Delta_2 \;\cup\; \ldots \\
\text{wobei:} & \Delta_{i+1} = & F\,(F^i(\emptyset))\backslash F^i(\emptyset) \\
& = & F\,(\Delta_i)\backslash(\Delta_1 \cup \ldots \cup \Delta_i) \quad \text{mit} \;\; \Delta_0 = \emptyset
\end{array}
$$

Nehmen wir an, die Kosten von $F\ x$ seien $1 + \#x$.

Dann summieren sich die Kosten zu:

naiv	$1 + 2 + \ldots + n + n \quad = \quad \frac{1}{2}n(n+3)$
semi-naiv	$2n$

wobei n die Kardinalität des Ergebnisses ist.

$\Longrightarrow$ Man spart einen linearen Faktor :-)

2.2 Peephole Optimierung

Idee:

- Schiebe ein kleines Fenster über das Programm.
- Optimiere agressiv innerhalb des Fensters. D.h.:

  $\longrightarrow$ Beseitige Redundanzen!

  $\longrightarrow$ Ersetze innerhalb des Fensters teure Operationen durch billige!

Beispiele:

$$x = x + 1; \quad \Longrightarrow \quad x{+}{+};$$

// sofern es dafür eine spezielle Instruktion gibt :-)

$$z = y - a + a; \quad \Longrightarrow \quad z = y;$$

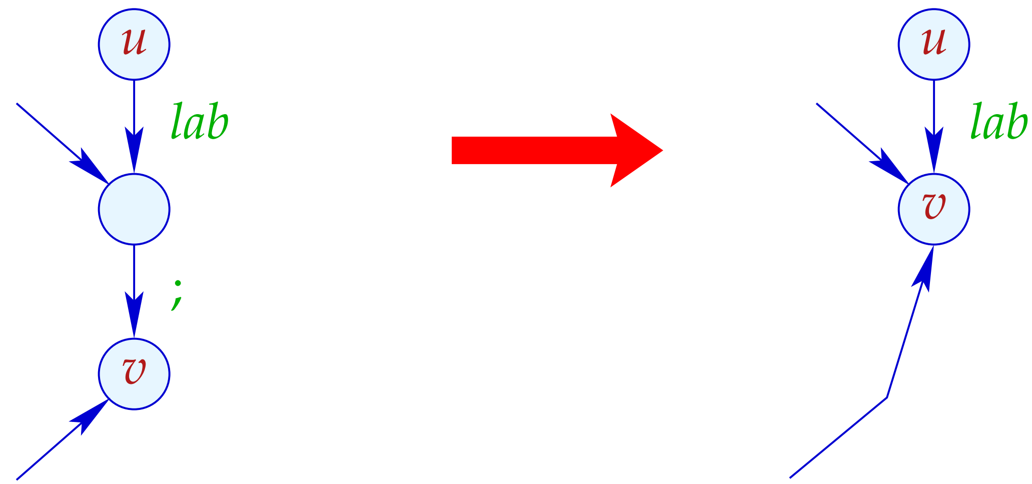
// algebraische Umformungen :-)

$$x = x; \quad \Longrightarrow \quad ;$$

$$x = 0; \quad \Longrightarrow \quad x = x \oplus x;$$

$$x = 2 \cdot x; \quad \Longrightarrow \quad x = x + x;$$

Wichtiges Teilproblem: *nop*-Optimierung



→ Ist $(v_1, ;, v)$ eine Kante, hat v_1 keine weitere ausgehende Kante.

→ Folglich dürfen wir v_1 und v identifizieren :-)

→ Die Reihenfolge der Identifizierungen ist egal :-))

Implementierung:

- Wir konstruieren eine Funktion $\mathsf{next} : \mathit{Nodes} \to \mathit{Nodes}$ mit:

$$\mathsf{next}\ u = \begin{cases} \mathsf{next}\ v & \text{falls} \quad (u, ;, v) \quad \text{Kante} \\ u & \text{sonst} \end{cases}$$

  Achtung: Diese Definition ist nur rekursiv, wenn es ;-Schleifen gibt ???

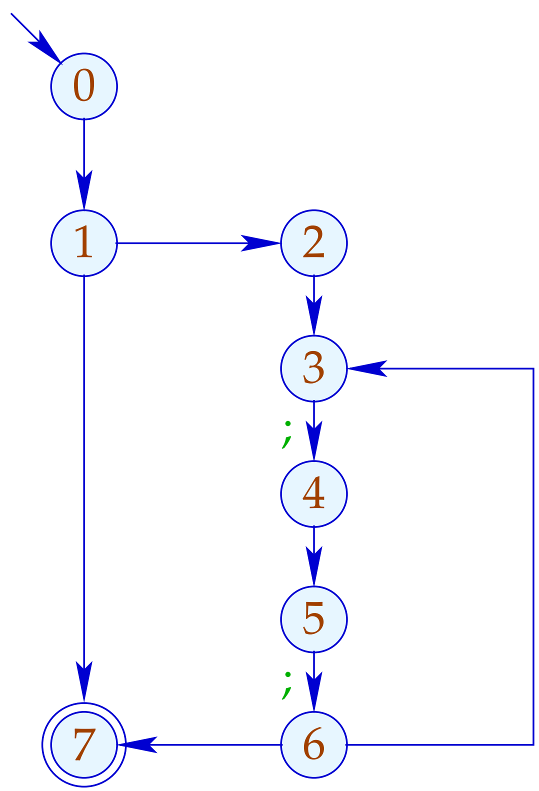
- Wir ersetzen jede Kante:

$$(u, \mathit{lab}, v) \qquad \Longrightarrow \qquad (u, \mathit{lab}, \mathsf{next}\ v)$$

... sofern $\mathit{lab} \neq ;$

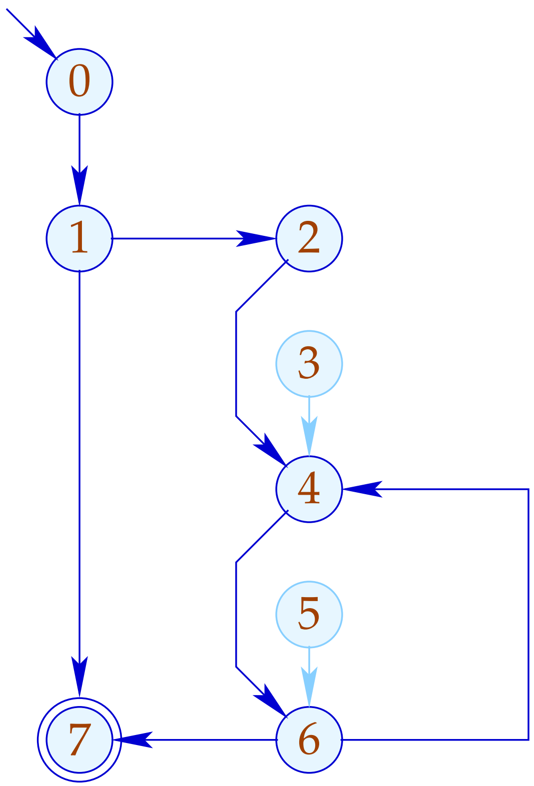
- Alle ;-Kanten werfen wir weg ;-)

Beispiel:



$$\text{next } 1 = 1$$
$$\text{next } 3 = 4$$
$$\text{next } 5 = 6$$

Beispiel:



$$\text{next } 1 = 1$$

$$\text{next } 3 = 4$$

$$\text{next } 5 = 6$$

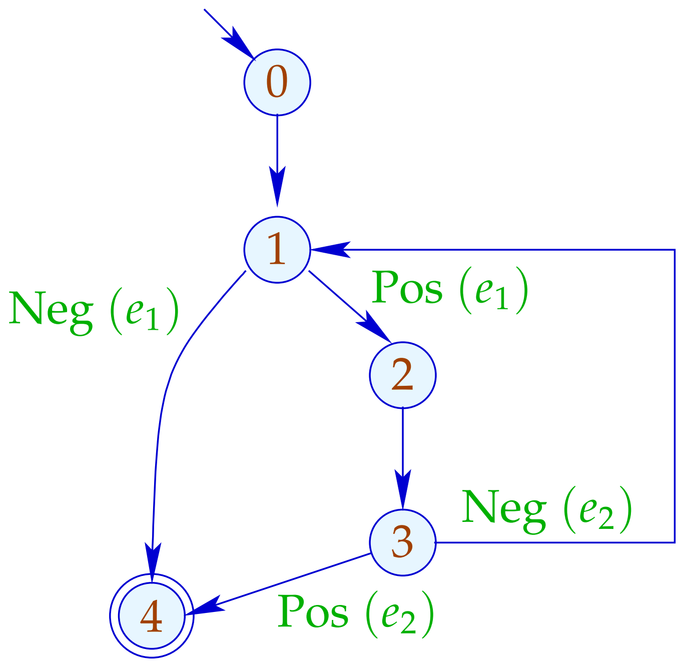
2. Teilproblem: Linearisierung

Der CFG muss nach der Optimierung wieder in eine lineare Abfolge von Instruktionen gebracht werden :-)

Achtung:

Nicht jede Linearisierung ist gleich gut !!!

Beispiel:



0:

1: if (e_1) goto 2;

4: halt

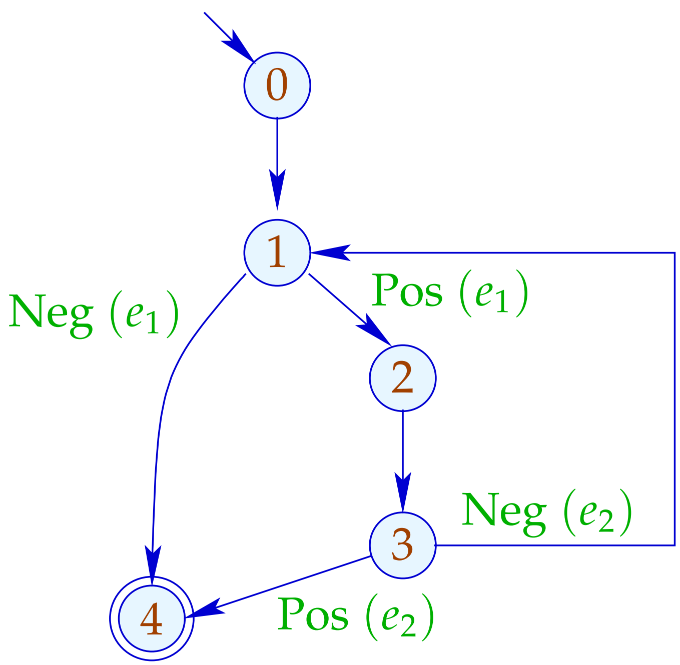
2: $\boxed{\text{Rumpf}}$

3: if (e_2) goto 4;

goto 1;

Schlecht: Der Schleifen-Rumpf wird angesprungen :-(

Beispiel:



0:

1: if $(!e_1)$ goto 4;

2: $\boxed{\text{Rumpf}}$

3: if $(!e_2)$ goto 1;

4: halt

// besseres Cache-Verhalten :-)

Idee:

- Gib jedem Knoten eine Temperatur!
- Springe stets zu

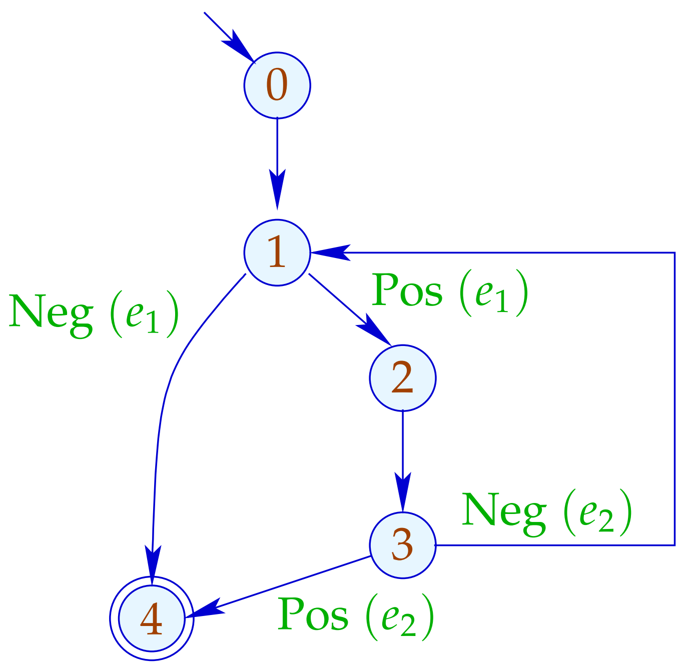
  (1) bereits behandelten Knoten;

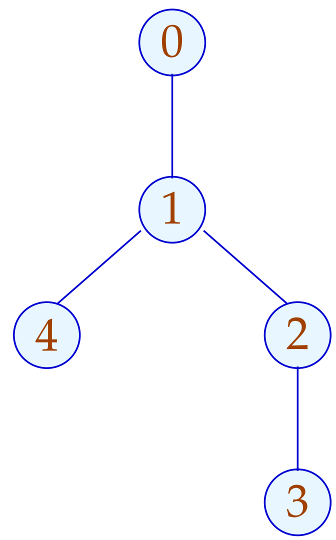
  (2) kälteren Knoten.

- Temperatur $\approx$ Schachtelungstiefe

Zur Berechnung benutzen wir den Prädominator-Baum und starke Zusammenhangskomponenten ...

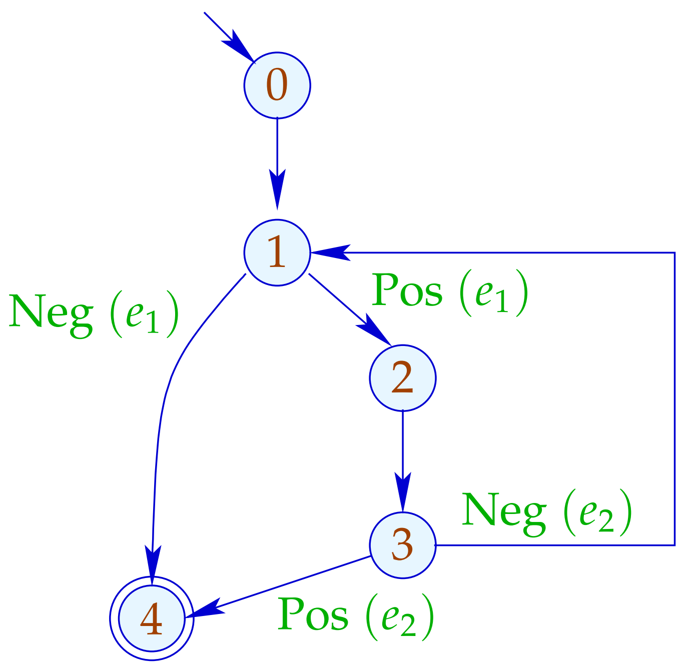
... im Beispiel:

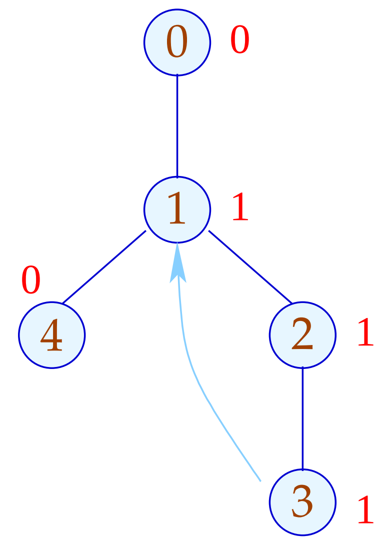




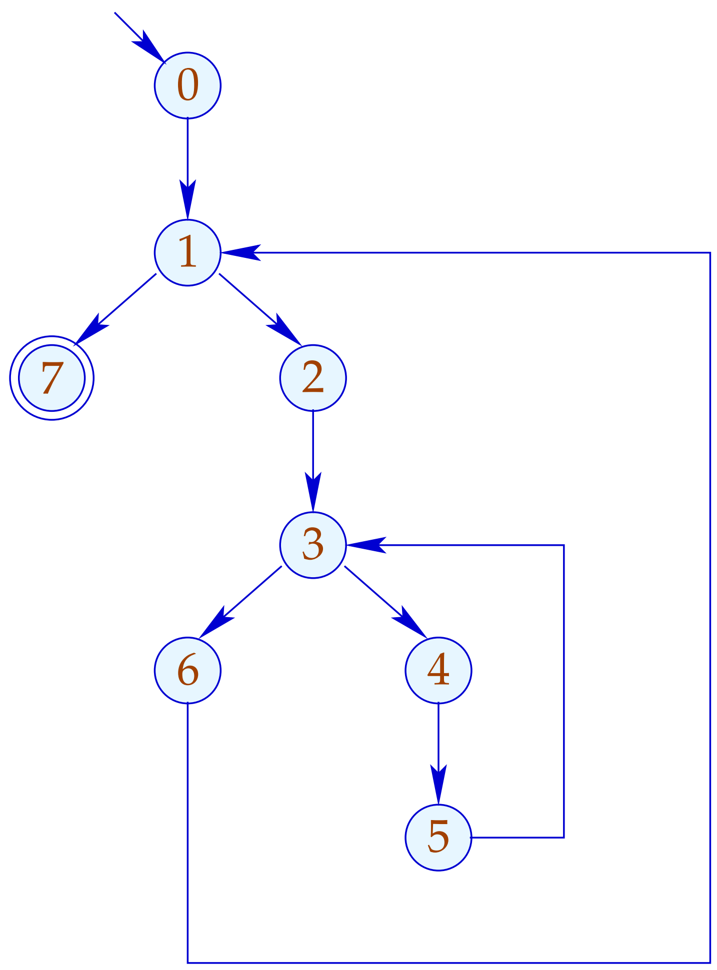
Der Teilbaum mit Rücksprung ist heißer ...

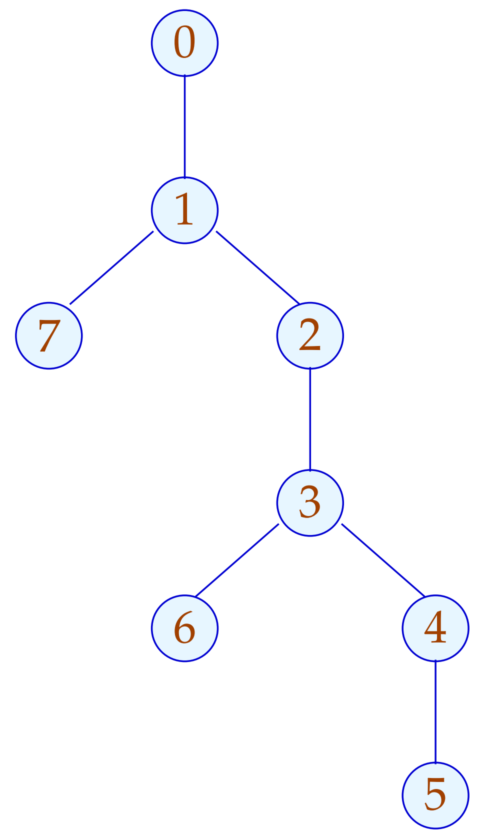
... im Beispiel:



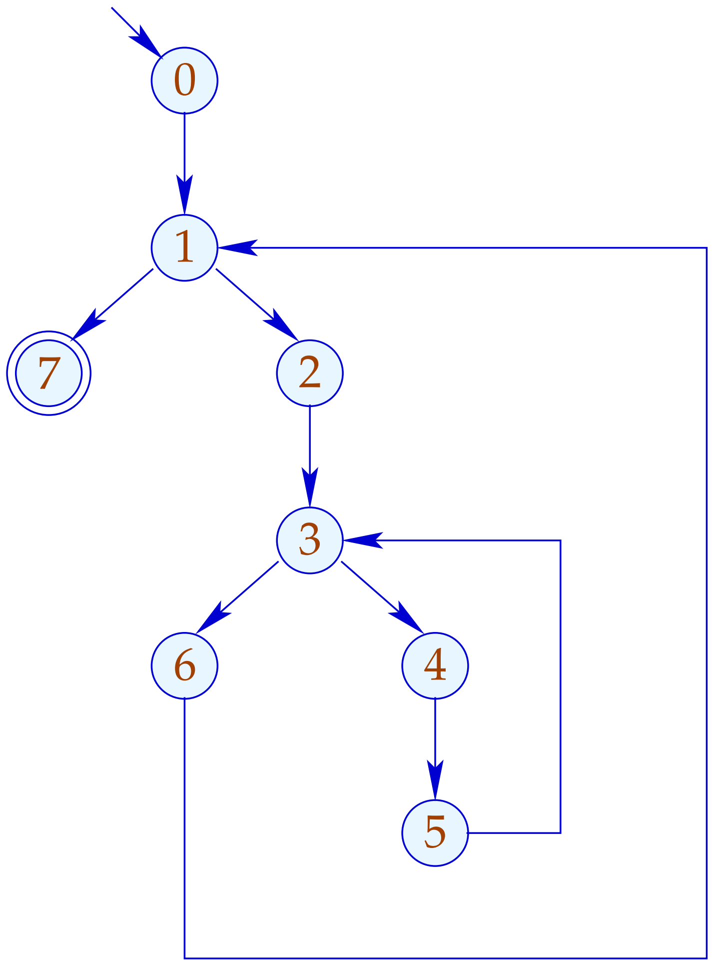


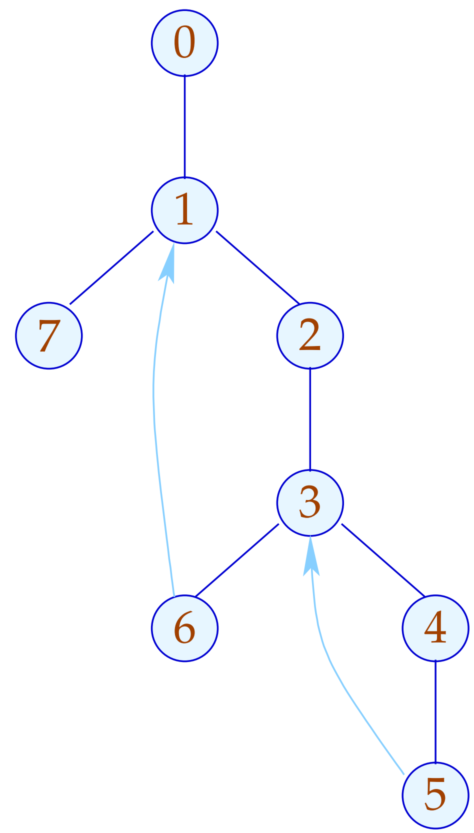
Komplizierteres Beispiel:



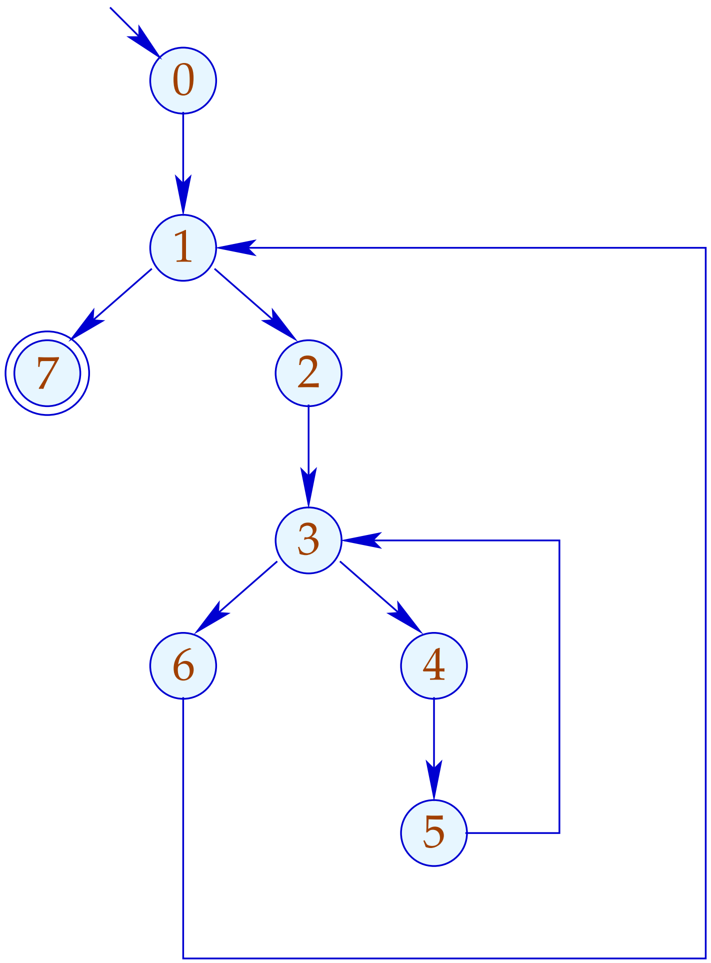


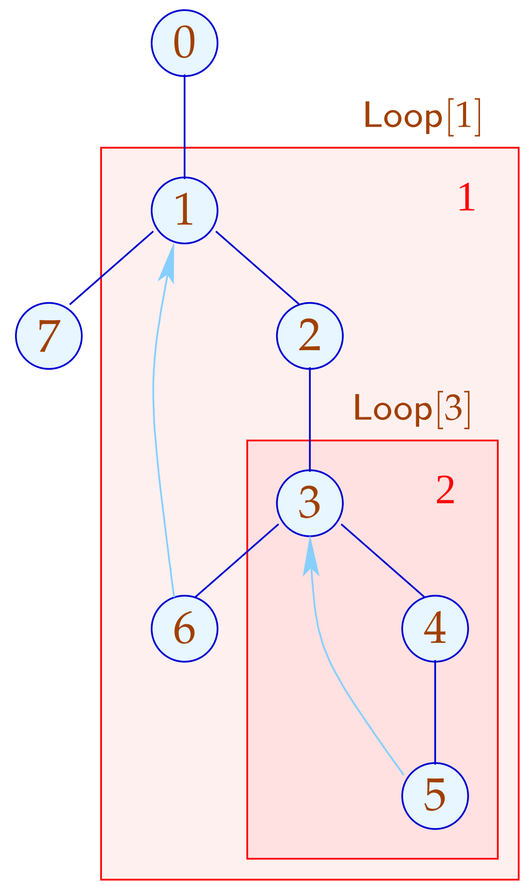
Komplizierteres Beispiel:





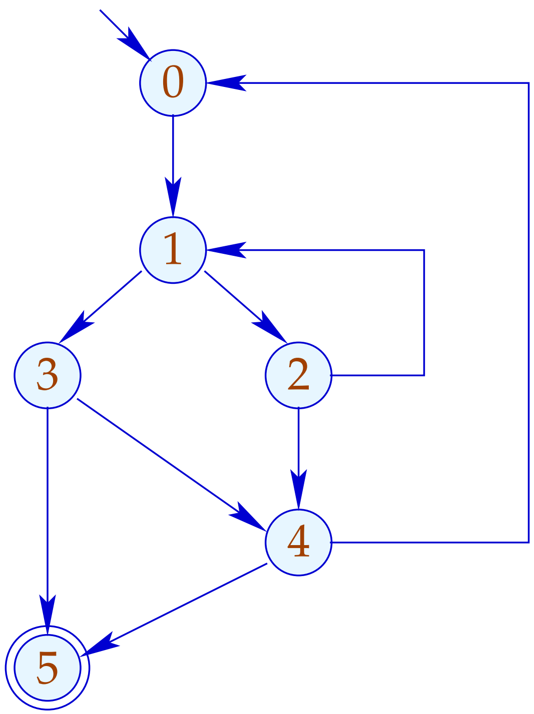
Komplizierteres Beispiel:

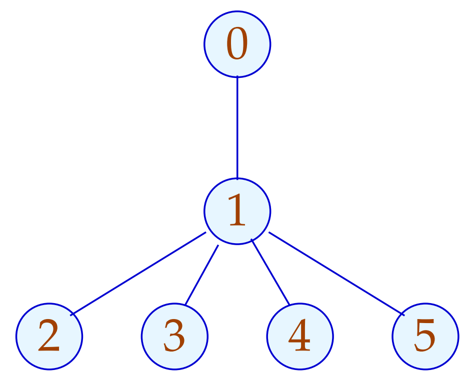




Unsere Definition von Loop sorgt dafür, dass (erkannte) Schleifen geschachtelt auftreten :-)

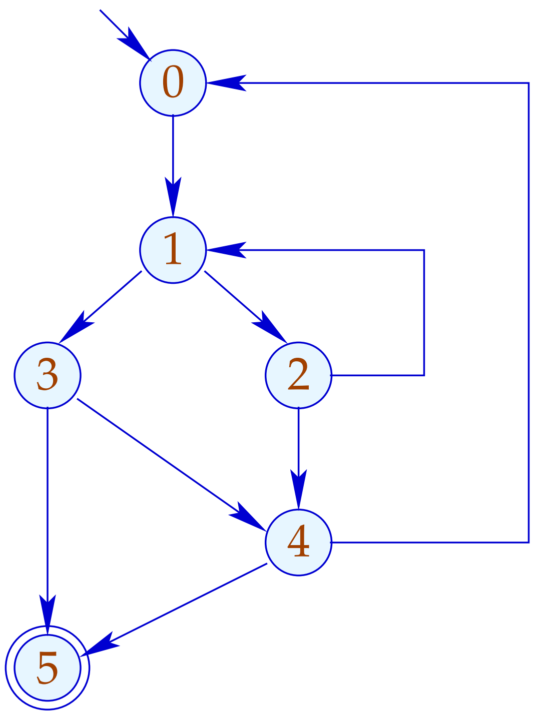
Sie ist auch für do-while-Schleifen mit breaks vernünftig...

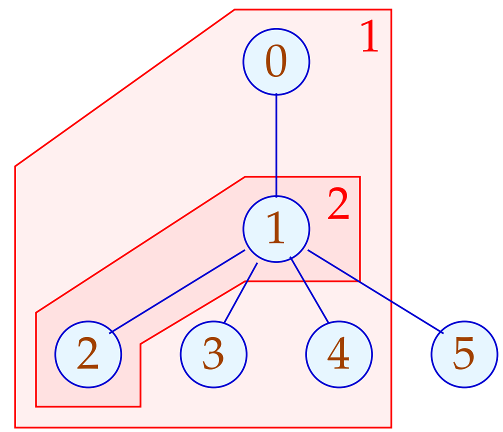




Unsere Definition von Loop sorgt dafür, dass (erkannte) Schleifen geschachtelt auftreten :-)

Sie ist auch für do-while-Schleifen mit breaks vernünftig...





Zusammenfassung: Das Verfahren

(1) Ermittlung einer Temperatur für jeden Knoten;

(2) Prä-order-DFS über den CFG;

- $\longrightarrow$ Führt eine Kante zu einem Knoten, für den wir bereits Code erzeugt haben, fügen wir einen Sprung ein.
- $\longrightarrow$ Hat ein Knoten zwei Nachfolger unterschiedlicher Temperatur, fügen wir einen Sprung zum kälteren der beiden ein.
- $\longrightarrow$ Hat ein Knoten zwei gleich warme Nachfolger, ist es egal ;-)

2.3 Funktionen

Wir erweitern unsere Mini-Programmiersprache um Funktionen und Funktions-Aufrufe. Dazu führen wir neue Statements ein:

$$\mathsf{ret} = f(b_1, \ldots, b_k); \qquad \mathsf{return}\; e;$$

Jede Funktion f besitzt eine Definition:

$$f\;(a_1, \ldots, a_n)\;\;\{\;\; stmt^* \;\;\}$$

// a_i formale Parameter

// b_i aktuelle Parameter

// ret Register für Rückgabewert

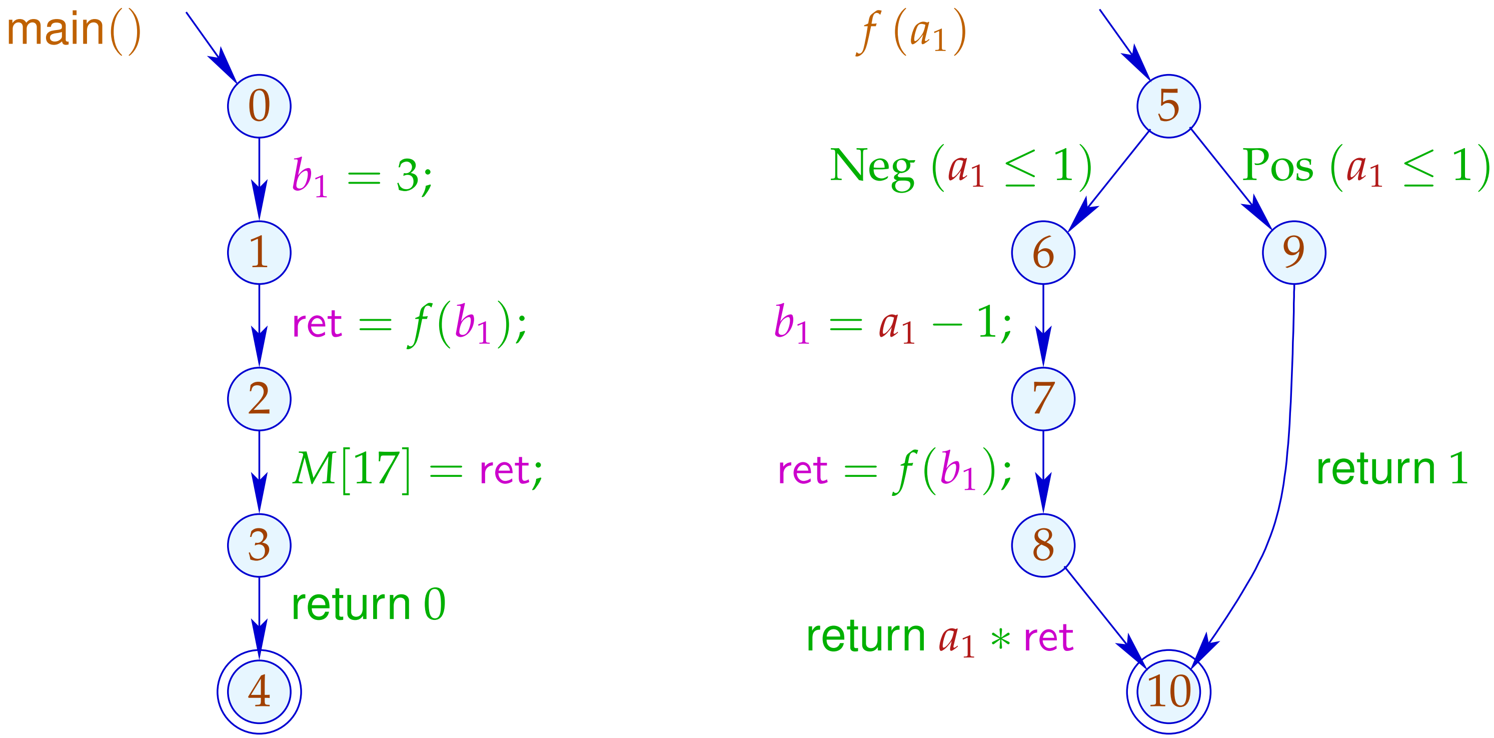
Die Programm-Ausführung startet mit dem Aufruf einer (parameterlosen) Funktion $\mathsf{main}\,()$.

Beispiel:

$$
\begin{array}{ll}
\text{main}\,() \;\; \{ & f\,(a_1) \;\; \{ \\
\qquad b_1 = 3; & \qquad \text{if } (a_1 \leq 1) \text{ return } 1; \\
\qquad \text{ret} = f(b_1); & \qquad\quad b_1 = a_1 - 1; \\
\qquad M[17] = \text{ret}; & \qquad \text{ret} = f(b_1); \\
\qquad \text{return } \; 0; & \qquad \text{return } \; a_1 \cdot \text{ret}; \\
\} & \}
\end{array}
$$

Solche Programme lassen sich durch eine Menge von CFGs darstellen: einem für jede Funktion ...

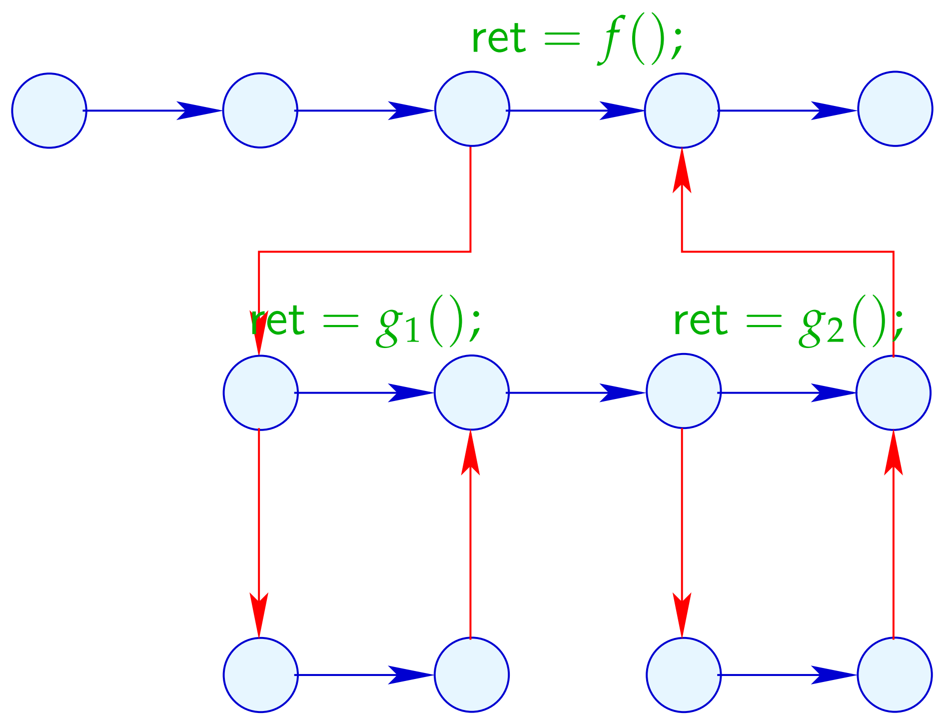
... im Beispiel:



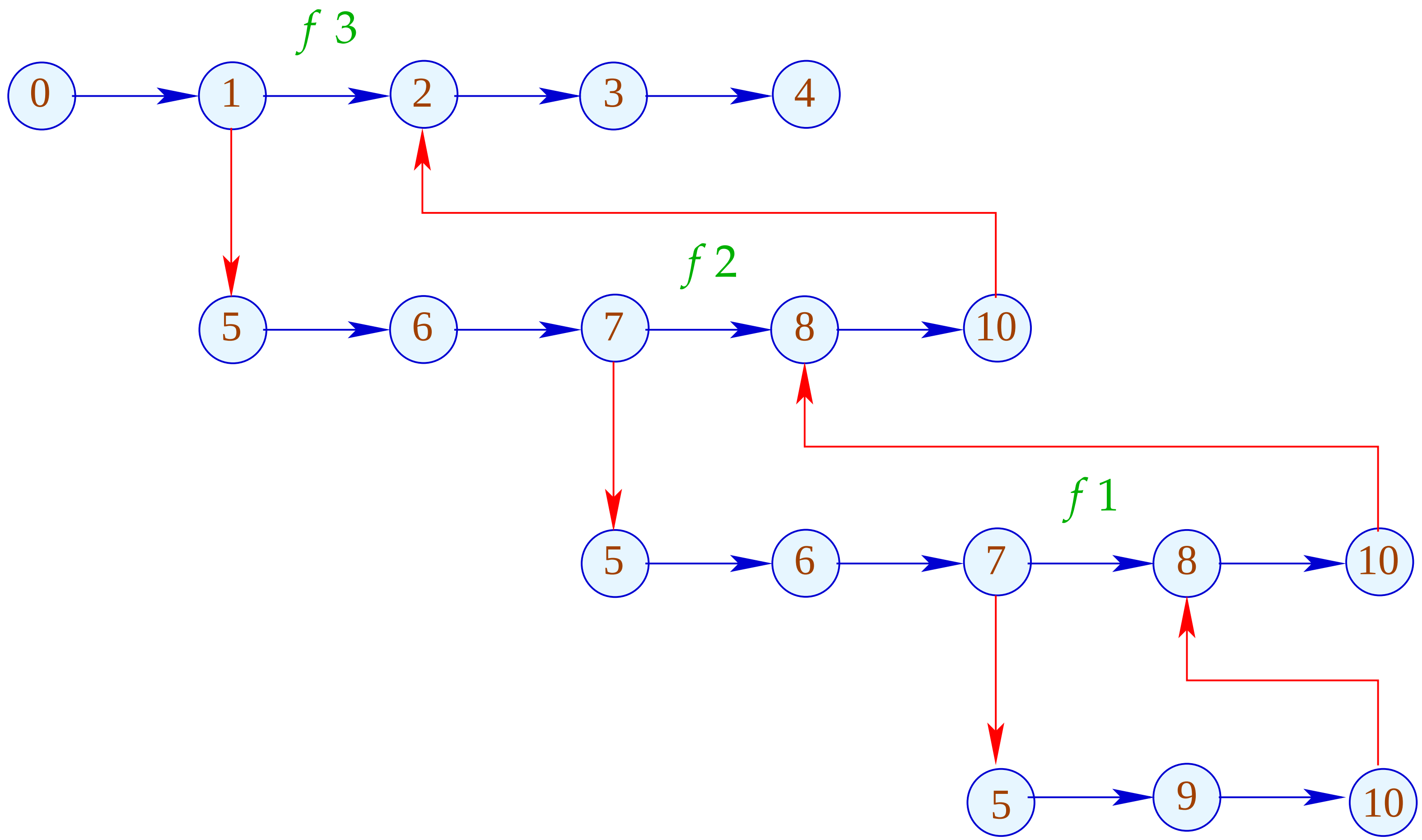
main()
0
$b_1 = 3;$
1
$\text{ret} = f(b_1);$
2
$M[17] = \text{ret};$
3
return 0
4
$f(a_1)$
5
$\text{Neg}(a_1 \leq 1)$
$\text{Pos}(a_1 \leq 1)$
6
9
$b_1 = a_1 - 1;$
7
$\text{ret} = f(b_1);$
return 1
8
$\text{return } a_1 * \text{ret}$
10

Um solche Programme zu optimieren, benötigen wir eine erweiterte operationelle Semantik ;-)

Programm-Ausführungen sind nicht mehr Pfade, sondern Wälder:



... im Beispiel:



Die Funktion $[\![.]\!]$ erweitern wir auf Berechnungs-Wälder w :

$$[\![w]\!] : (\mathit{Vars} \to \mathbb{Z}) \times (\mathbb{N} \to \mathbb{Z}) \to (\mathit{Vars} \to \mathbb{Z}) \times (\mathbb{N} \to \mathbb{Z})$$

Für einen Aufruf $k = (u, \mathsf{ret} = f\,(b_1, \ldots, b_k);, v)$ müssen wir:

- die Anfangwerte der lokalen Variablen ermitteln:

$$\mathsf{enter}_k\ \rho\ x = \begin{cases} \rho\, b_i & \text{falls} \quad x = a_i \\ 0 & \text{sonst} \end{cases}$$

- ... den berechneten Rückgabe-Wert in ret ablegen:

$$\mathsf{combine}\ (\rho_1, \rho_2) = \rho_1 \oplus \{\mathsf{ret} \to \rho_2\ \mathsf{ret}\}$$

- ... dazwischen den Berechnungs-Wald der Funktion auswerten:

$$
\begin{aligned}
[\![k\ \langle w \rangle]\!]\ (\rho, \mu) \quad = & \\
& \mathsf{let}\ \ (\rho_1, \mu_1) = [\![w]\!]\ (\mathsf{enter}_k\ \rho, \mu) \\
& \mathsf{in}\quad (\mathsf{combine}\ (\rho, \rho_1), \mu_1)
\end{aligned}
$$

Ein Return $\quad k = (u, \mathsf{return}\ \ e;, v) \quad$ ist eine Zuweisung an $\quad \mathsf{ret}$:

$$
[\![k]\!]\ (\rho, \mu) \quad = \quad (\rho \oplus \{\mathsf{ret} \mapsto [\![e]\!]\ \rho\}, \mu)
$$

Achtung:

- $[\![w]\!]$ ist i.a. nur partiell definiert :-)
- Die Benutzung von speziellen Registern a_i, b_i, ret repräsentiert eine bestimmte Aufruf-Konvention.
- Die normale operationelle Semantik arbeitet mit Konfigurationen, die Aufrufkeller verwalten.
- Berechnungs-Wälder eignen sich aber besser zur Konstruktion von Analysen und Korrektheitsbeweisen :-)
- Es ist eine lästige (aber nützliche) Aufgabe, die Äquivalenz der beiden Ansätze zu zeigen ...

Konfigurationen:

$$\begin{array}{lll} \mathit{configuration} & == & \mathit{stack} \times \mathit{store} \\ \mathit{store} & == & \mathbb{N} \to \mathbb{Z} \\ \mathit{stack} & == & \mathit{frame} \cdot \mathit{frame}^* \\ \mathit{frame} & == & \mathit{point} \times \mathit{locals} \\ \mathit{locals} & == & (\mathit{Vars} \to \mathbb{Z}) \end{array}$$

Ein *frame* (Kellerrahmen) beschreibt den lokalen Berechnungszustand innerhalb eines Funktionsaufrufs :-)

Den Rahmen des aktuellen Aufrufs schreiben wir links.

Berechnungsschritte beziehen sich auf den aktuellen Aufruf :-)

Zusätzlich benötigte Arten von Schritten:

Aufruf $k = (u, \mathsf{ret} = f\,(b_1, \ldots, b_k);, v)$:

$$(\boxed{(u, \rho)} \cdot \sigma, \mu) \quad \Longrightarrow \quad (\boxed{(u_f, \mathsf{enter}_k\, \rho) \cdot (v, \rho)} \cdot \sigma, \mu)$$

u_f Anfangspunkt von f

Rückkehr:

$$(\boxed{(r_f, \rho_2) \cdot (v, \rho_1)} \cdot \sigma, \mu) \quad \Longrightarrow \quad (\boxed{(v, \mathsf{combine}\,(\rho_1, \rho_2))} \cdot \sigma, \mu)$$

r_f Endpunkt von f

Rückgabe: $k = (u, \mathsf{return}\ \ e;, v)$:

$$(\boxed{(u, \rho)} \cdot \sigma, \mu) \quad \Longrightarrow \quad (\boxed{(v, \rho \oplus \{\mathsf{ret} \mapsto [\![e]\!]\, \rho\})} \cdot \sigma, \mu)$$

Mit dem Aufruf-Keller verwalten wir explizit den DFS-Durchlauf über den Berechnungswald :-)

... im Beispiel:

1	$b_1 \mapsto 3$

Mit dem Aufruf-Keller verwalten wir explizit den DFS-Durchlauf über den Berechnungswald :-)

... im Beispiel:

5	$a_1 \mapsto 3$
2	

Mit dem Aufruf-Keller verwalten wir explizit den DFS-Durchlauf über den Berechnungswald :-)

... im Beispiel:

7	$a_1 \mapsto 3$
2	

Mit dem Aufruf-Keller verwalten wir explizit den DFS-Durchlauf über den Berechnungswald :-)

... im Beispiel:

5	$a_1 \mapsto 2$
8	$a_1 \mapsto 3$
2	

Mit dem Aufruf-Keller verwalten wir explizit den DFS-Durchlauf über den Berechnungswald :-)

... im Beispiel:

7	$a_1 \mapsto 2$
8	$a_1 \mapsto 3$
2	

Mit dem Aufruf-Keller verwalten wir explizit den DFS-Durchlauf über den Berechnungswald :-)

... im Beispiel:

5	$a_1 \mapsto 1$
8	$a_1 \mapsto 2$
8	$a_1 \mapsto 3$
2	

Mit dem Aufruf-Keller verwalten wir explizit den DFS-Durchlauf über den Berechnungswald :-)

... im Beispiel:

10	$\mathsf{ret} \mapsto 1$
8	$a_1 \mapsto 2$
8	$a_1 \mapsto 3$
2	

Mit dem Aufruf-Keller verwalten wir explizit den DFS-Durchlauf über den Berechnungswald :-)

... im Beispiel:

8	$\mathsf{ret} \longmapsto 1$
8	$a_1 \longmapsto 3$
2	

Mit dem Aufruf-Keller verwalten wir explizit den DFS-Durchlauf über den Berechnungswald :-)

... im Beispiel:

10	$\mathsf{ret} \mapsto 2$
8	$a_1 \mapsto 3$
2	

Mit dem Aufruf-Keller verwalten wir explizit den DFS-Durchlauf über den Berechnungswald :-)

... im Beispiel:

8	ret $\mapsto 2$
2	

Mit dem Aufruf-Keller verwalten wir explizit den DFS-Durchlauf über den Berechnungswald :-)

... im Beispiel:

10	ret $\mapsto$ 6
2	

Mit dem Aufruf-Keller verwalten wir explizit den DFS-Durchlauf über den Berechnungswald :-)

... im Beispiel:

2	ret $\mapsto$ 6