Berechnung:

Wir sammeln die Knoten entlang Pfaden auf mithilfe der Analyse:

$$\mathbb{P} = 2^{Nodes} \quad , \qquad \sqsubseteq \; = \; \supseteq$$

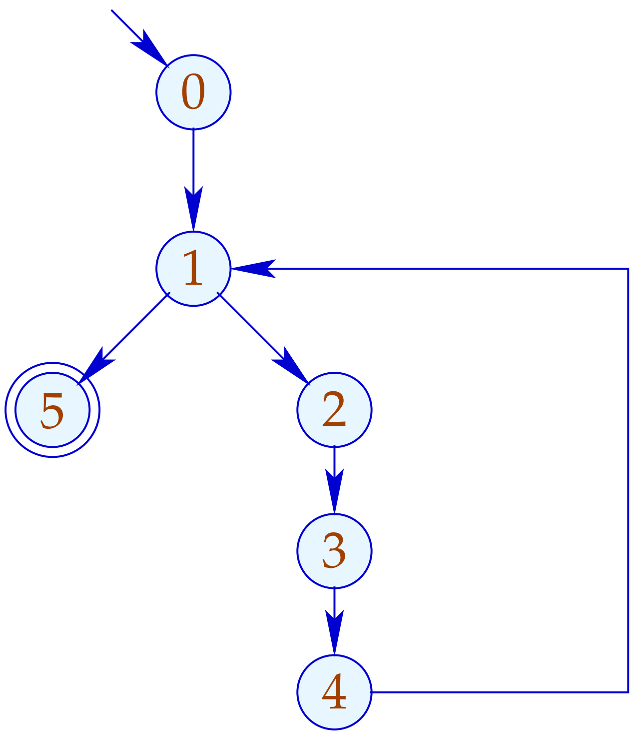
$$[\![(\_, \_, v)]\!]^\sharp \, P \;\; = \;\; P \cup \{v\}$$

Dann ist die Menge $\mathcal{P}[v]$ der Prädominatoren:

$$\mathcal{P}[v] = \bigcap \{ [\![\pi]\!]^\sharp \, \{start\} \mid \pi : start \rightarrow^* v \}$$

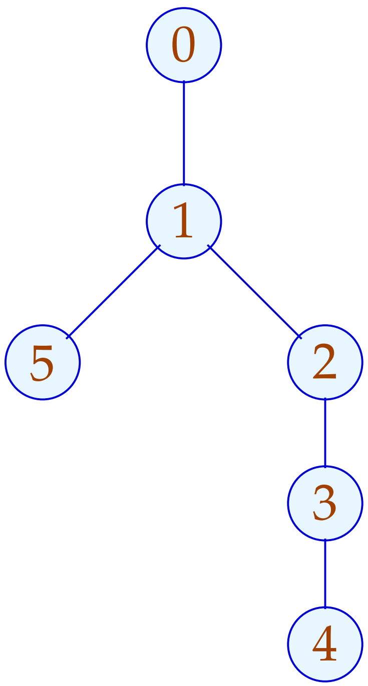
Da die $[\![k]\!]^\sharp$ distributiv sind, können wir die $\mathcal{P}[v]$ mithilfe von Fixpunkt-Iteration berechnen :-)

Beispiel:



	$\mathcal{P}$
0	$\{0\}$
1	$\{0,1\}$
2	$\{0,1,2\}$
3	$\{0,1,2,3\}$
4	$\{0,1,2,3,4\}$
5	$\{0,1,5\}$

Die partielle Ordnung "$\Rightarrow$" im Beispiel:



	$\mathcal{P}$
0	$\{0\}$
1	$\{0,1\}$
2	$\{0,1,2\}$
3	$\{0,1,2,3\}$
4	$\{0,1,2,3,4\}$
5	$\{0,1,5\}$

Offenbar ist das Ergebnis ein Baum :-)

Tatsächlich gilt:

Satz:

Jeder Punkt v hat maximal einen unmittelbaren Prädominator.

Beweis:

Annahme:

Es gäbe $u_1 \neq u_2$, die v unmittelbar prädominieren.

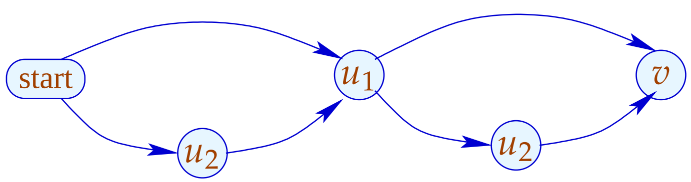
Gälte $u_1 \Rightarrow u_2$, wäre u_1 nicht unmittelbar.

Folglich müssen u_1, u_2 unvergleichbar sein :-)

Nun gilt für jedes $\pi : start \rightarrow^* v$:

$$\pi = \pi_1\ \pi_2 \qquad \text{mit} \qquad \begin{array}{l} \pi_1 : start \rightarrow^* u_1 \\ \pi_2 : u_1 \rightarrow^* v \end{array}$$

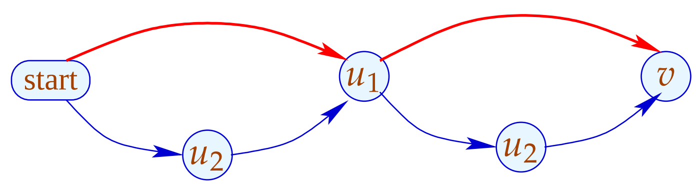
Sind u_1, u_2 aber unvergleichbar, gibt es einen Pfad: $start \rightarrow^* v$ ohne u_2 :



Nun gilt für jedes $\pi : start \longrightarrow^* v$:

$$\pi = \pi_1\ \pi_2 \qquad \text{mit} \qquad \begin{array}{l} \pi_1 : start \longrightarrow^* u_1 \\ \pi_2 : u_1 \longrightarrow^* v \end{array}$$

Sind u_1, u_2 aber unvergleichbar, gibt es einen Pfad: $start \longrightarrow^* v$ ohne u_2 :



Beobachtung:

Der Schleifenkopf einer while-Schleife dominiert jeden Knoten des Rumpfs.

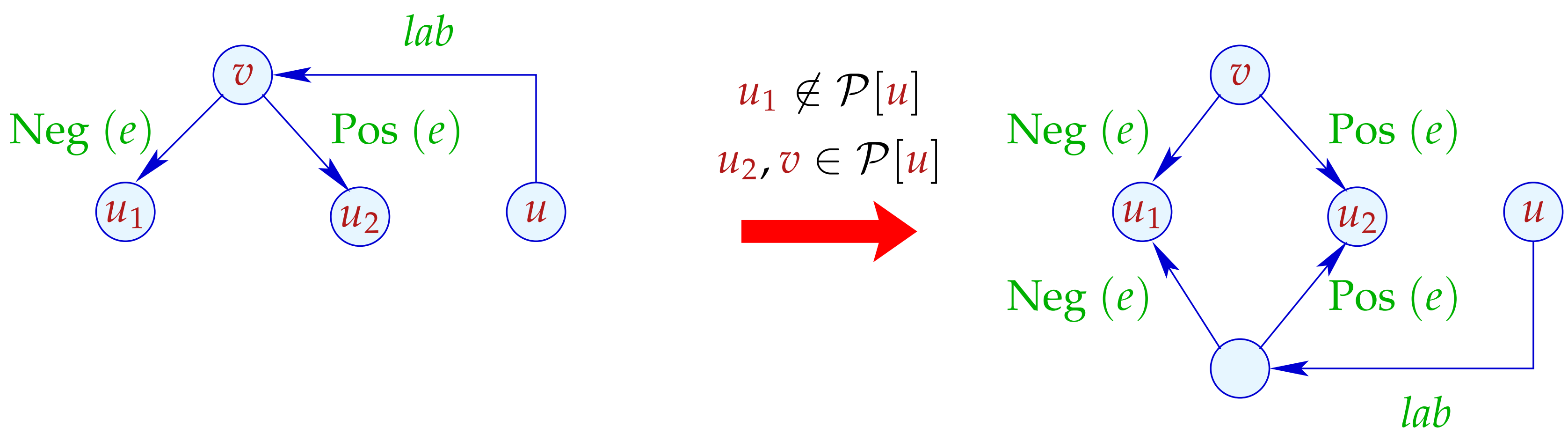
Einen Rücksprung vom Ende u zum Schleifenkopf v erkennt man daran, dass

$$v \in \mathcal{P}[u]$$

:-)

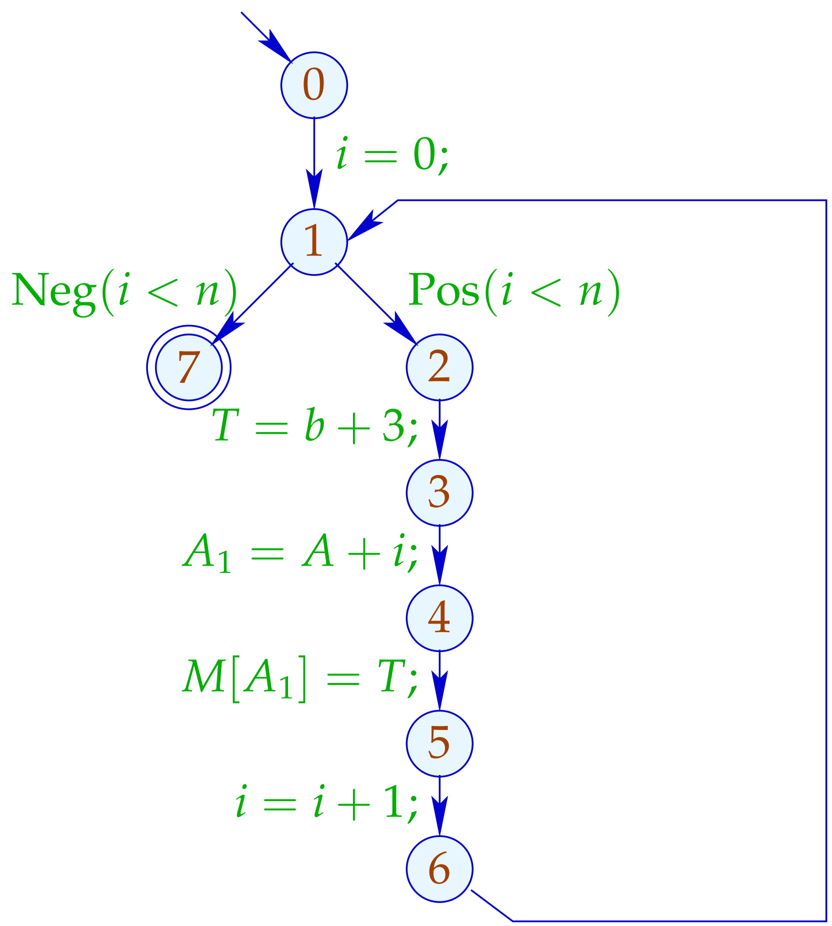
Damit definieren wir:

Transformation 7:

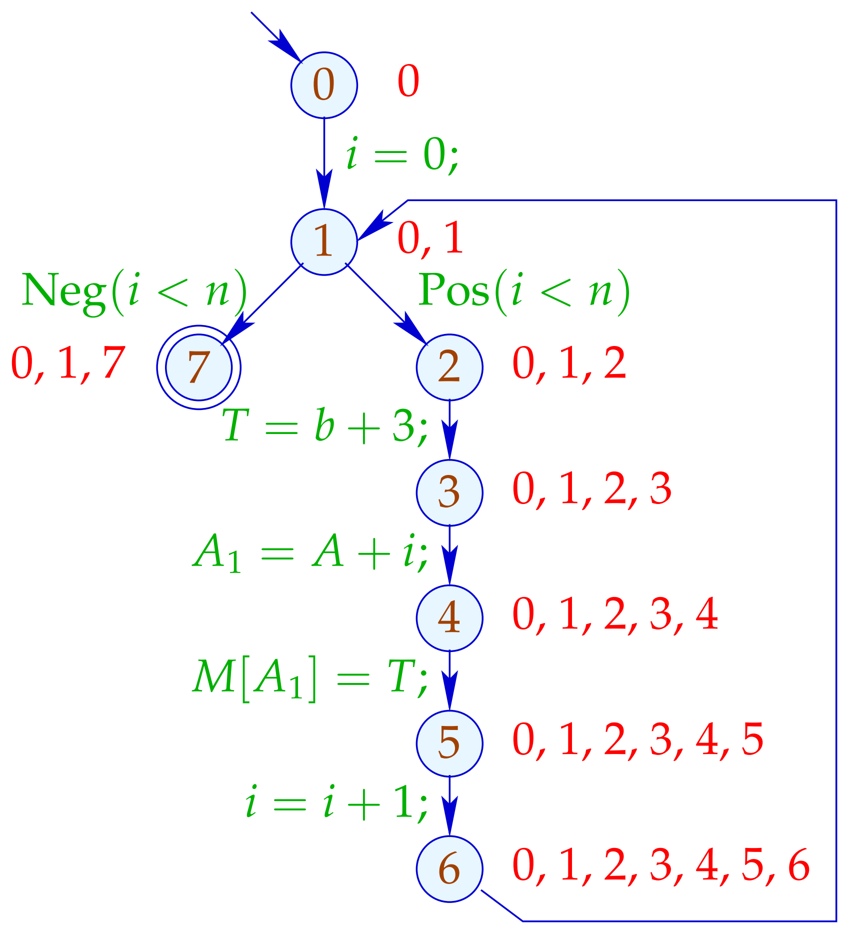


Wir duplizieren den Eintritts-Test an alle Rücksprung-Stellen :-)

... im Beispiel:

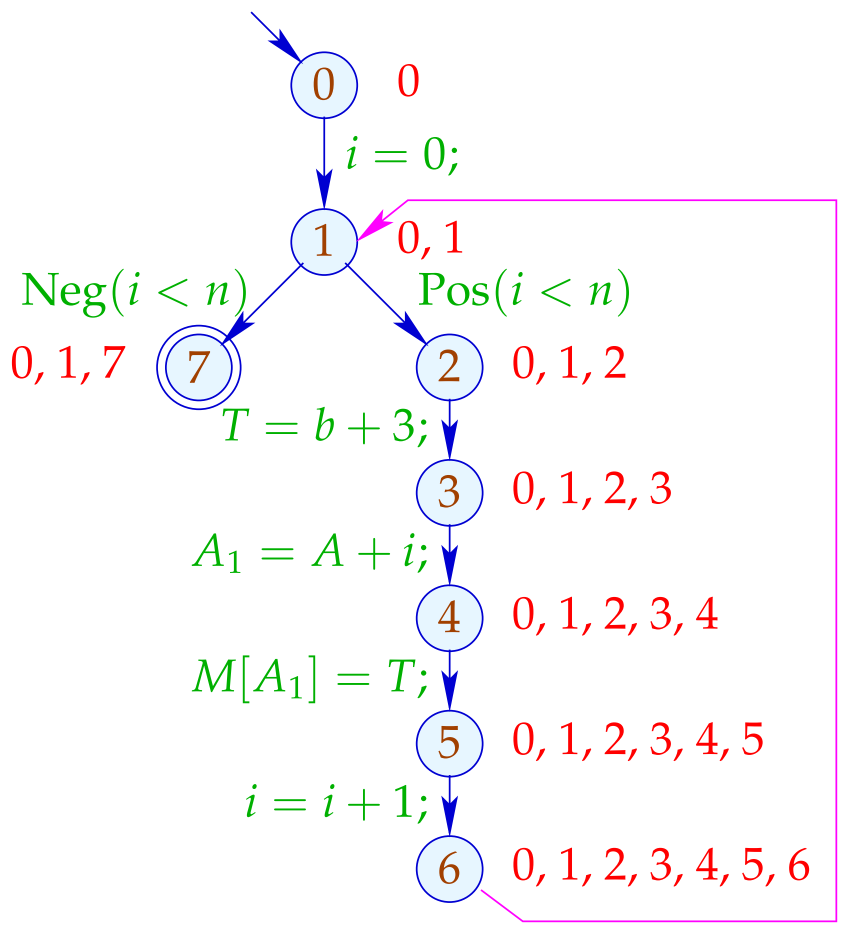


... im Beispiel:

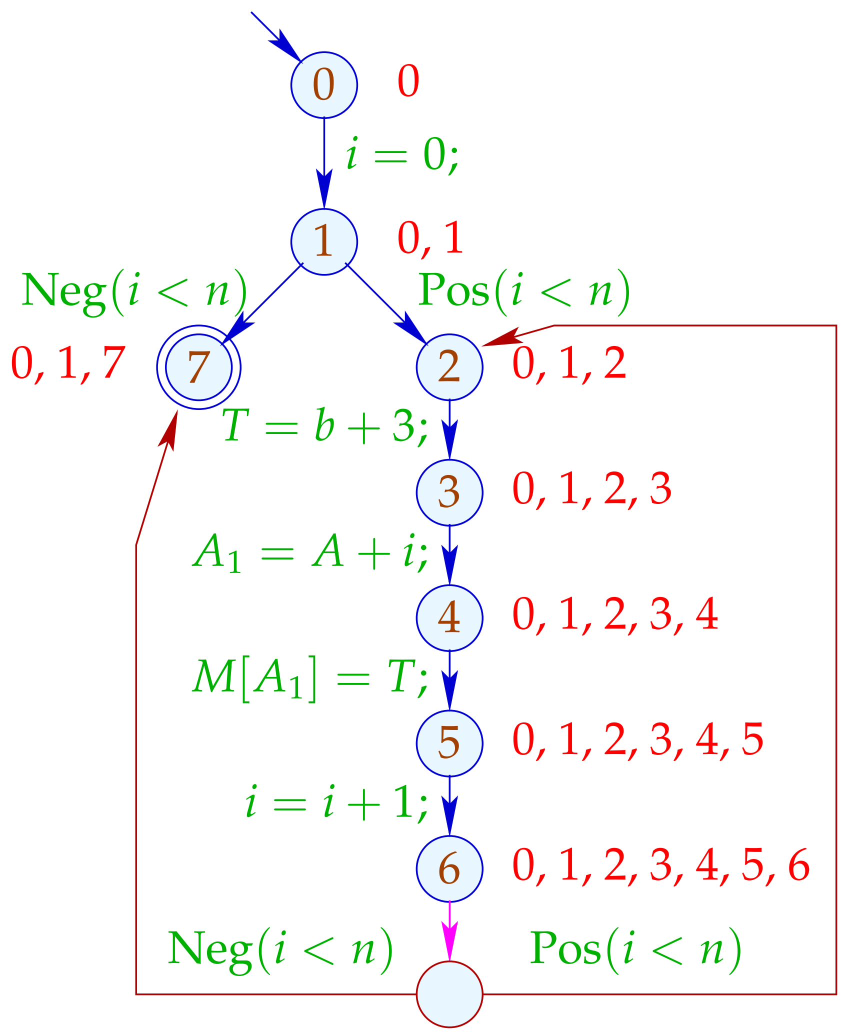


0
0
$i = 0;$
1
0, 1
Neg$(i < n)$
Pos$(i < n)$
7
0, 1, 7
2
0, 1, 2
$T = b + 3;$
3
0, 1, 2, 3
$A_1 = A + i;$
4
0, 1, 2, 3, 4
$M[A_1] = T;$
5
0, 1, 2, 3, 4, 5
$i = i + 1;$
6
0, 1, 2, 3, 4, 5, 6

... im Beispiel:


0
0
i = 0;
1
0, 1
Neg(i < n)
Pos(i < n)
7
0, 1, 7
2
0, 1, 2
T = b + 3;
3
0, 1, 2, 3
A1 = A + i;
4
0, 1, 2, 3, 4
M[A1] = T;
5
0, 1, 2, 3, 4, 5
i = i + 1;
6
0, 1, 2, 3, 4, 5, 6

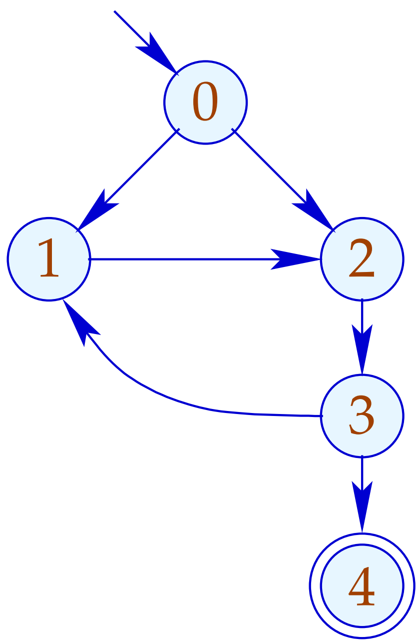
... im Beispiel:



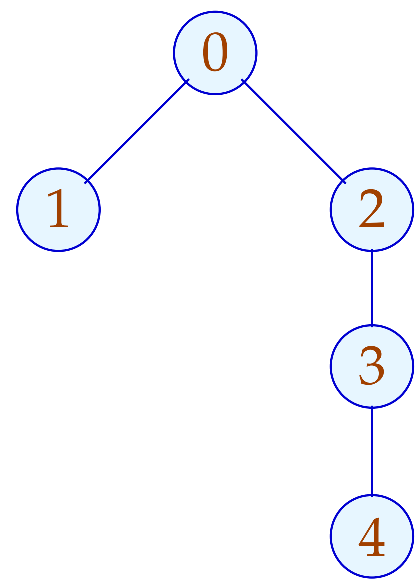
0
0
$i = 0;$
1
0, 1
$\text{Neg}(i < n)$
$\text{Pos}(i < n)$
0, 1, 7
7
2
0, 1, 2
$T = b + 3;$
3
0, 1, 2, 3
$A_1 = A + i;$
4
0, 1, 2, 3, 4
$M[A_1] = T;$
5
0, 1, 2, 3, 4, 5
$i = i + 1;$
6
0, 1, 2, 3, 4, 5, 6
$\text{Neg}(i < n)$
$\text{Pos}(i < n)$

Achtung:

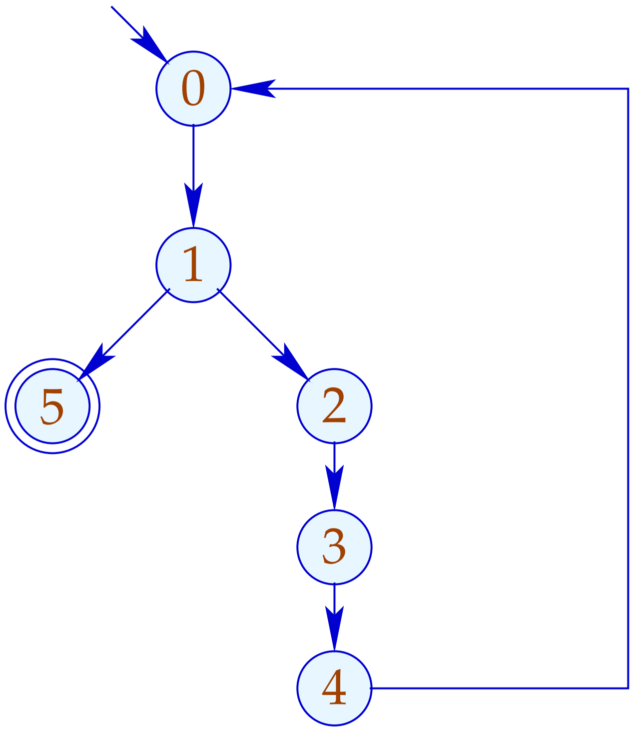
Es gibt ungewöhnliche Schleifen, die so nicht rotiert werden:

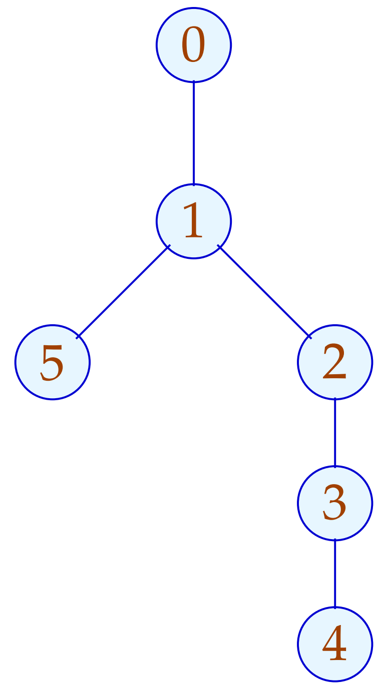


Prädominatoren:



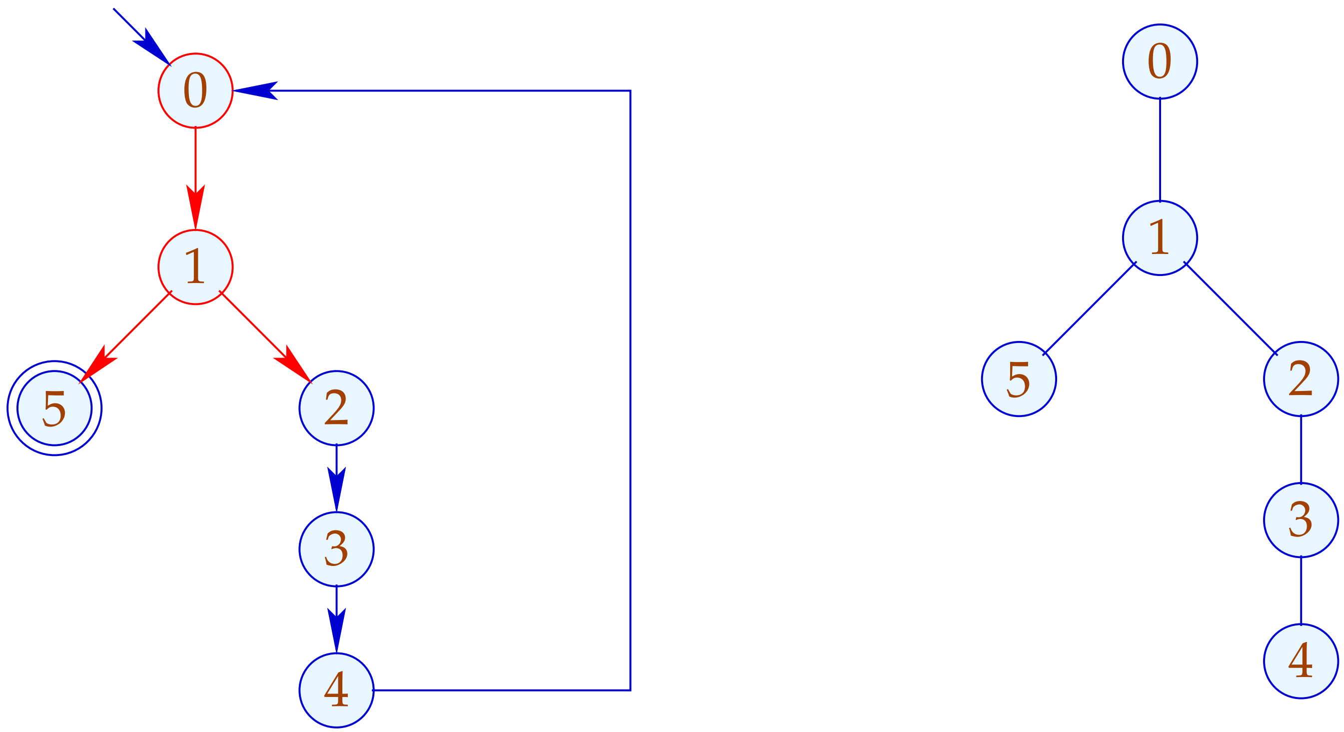
... leider aber auch gewöhnliche, die nicht rotiert werden:





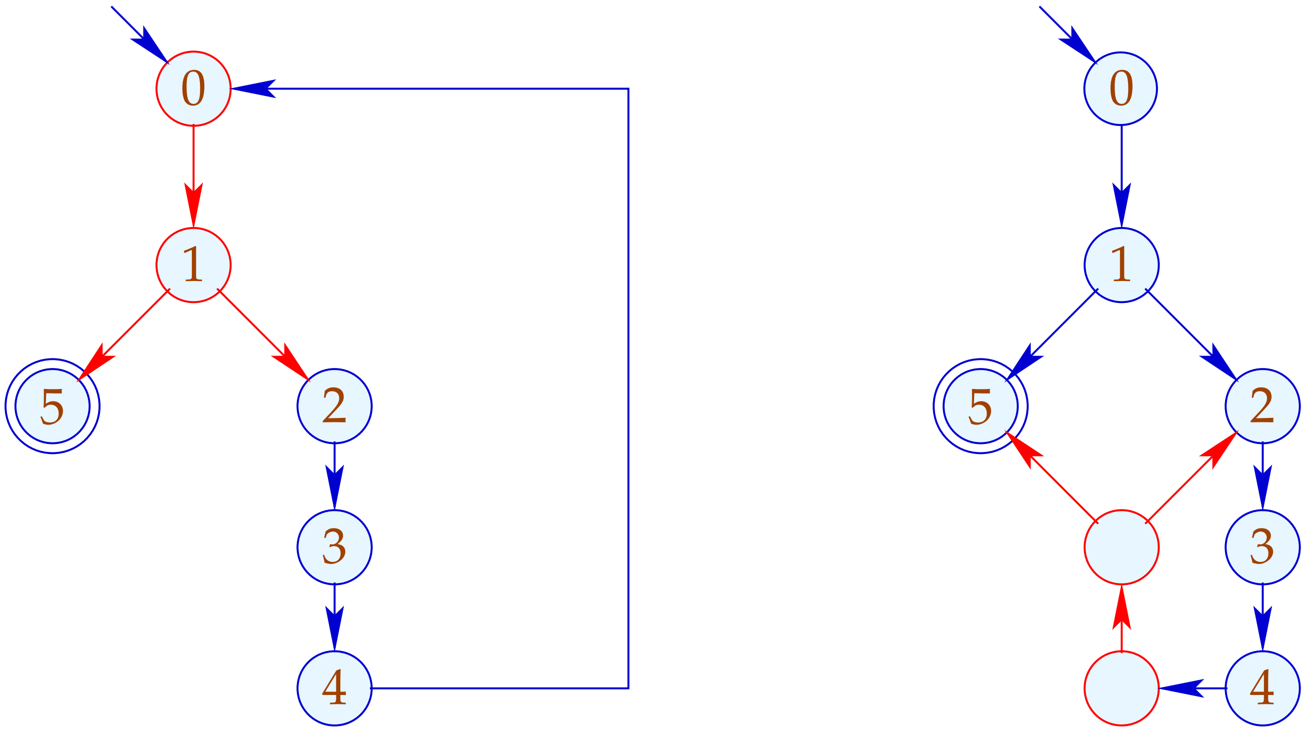
Hier müsste man den ganzen Pfad zwischen Rücksprung und Bedingung duplizieren :-(

... leider aber auch gewöhnliche, die nicht rotiert werden:



Hier müsste man den ganzen Pfad zwischen Rücksprung und Bedingung duplizieren :-(

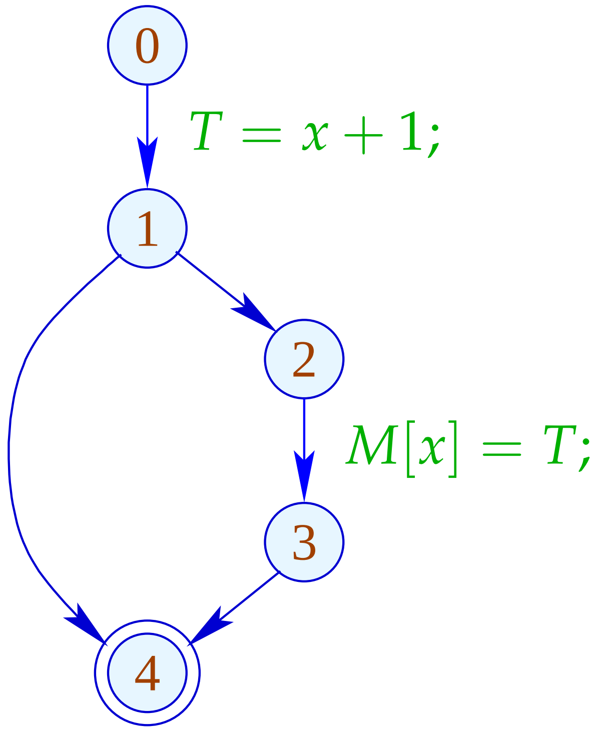
... leider aber auch gewöhnliche, die nicht rotiert werden:



Hier müsste man den ganzen Pfad zwischen Rücksprung und Bedingung duplizieren :-)

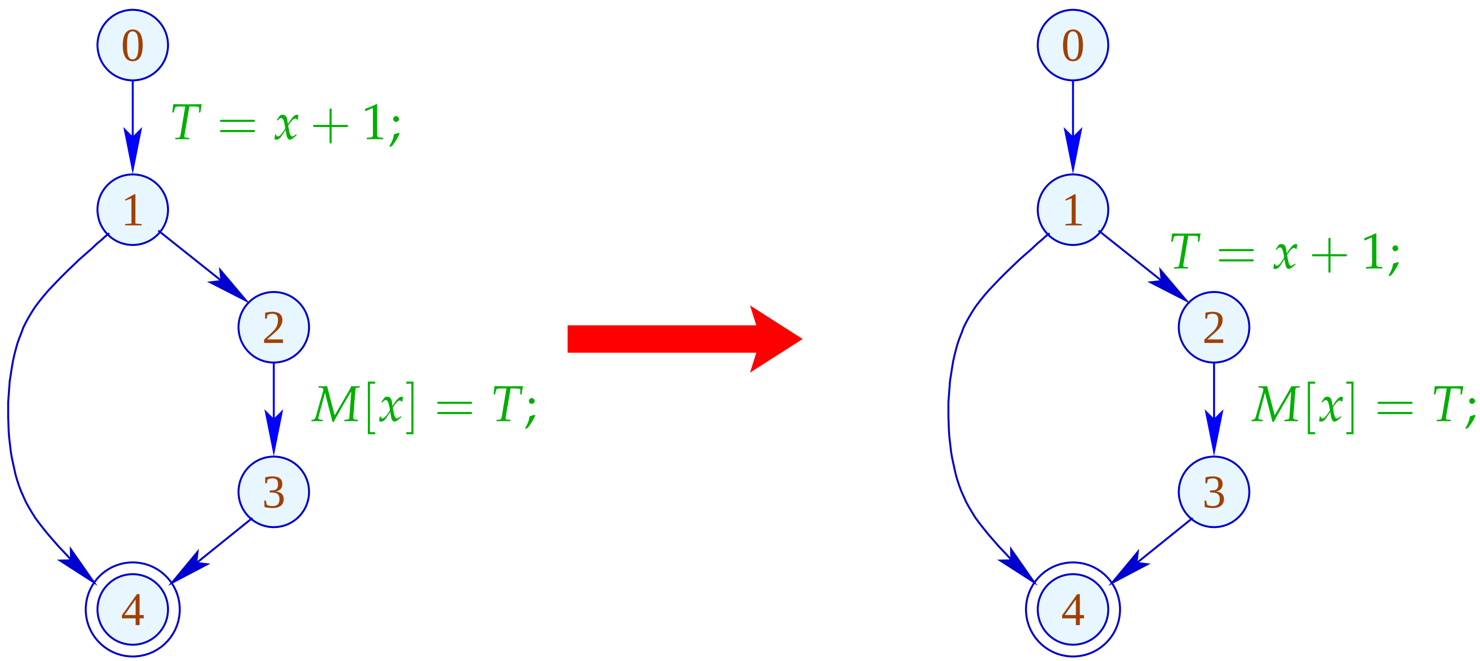
1.9 Beseitigung partiell toten Codes

Beispiel:



$x + 1$ muss nur auf einem der Pfade berechnet werden ;-(

Idee:



Problem:

- Die Definition $T_e = e;$ darf nur dorthin geschoben werden, wo sie eh verfügbar ist ;-)
- Die Definition muss weiterhin für Benutzungen zur Verfügung stehen ;-)

$$\Longrightarrow$$

Wir schieben sie an den Anfang einer Kante (u, lab, v) mit:

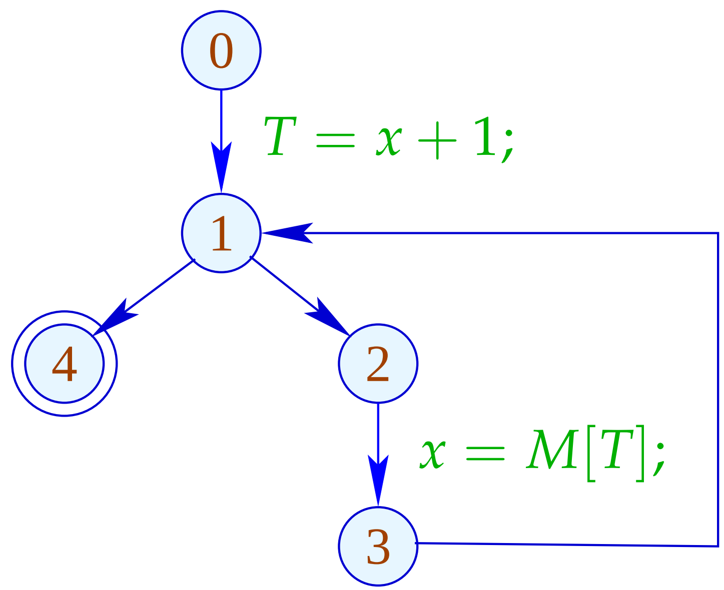
1. $e \in \mathcal{A}[u]$; und
2. $e \notin \mathcal{A}[v] \quad \vee \quad T_e \in Use(lab)$

Dabei ist:

lab	Use
$;$	$\emptyset$
$\text{Pos}(e)$	$Vars(e)$
$\text{Neg}(e)$	$Vars(e)$
$T_e = e;$	$Vars(e)$
$x = R;$	$\{R\}$
$x = M[R];$	$\{R\}$
$M[R] = x;$	$\{x, R\}$

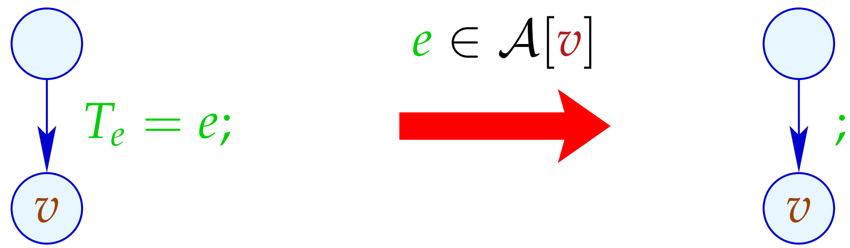
Achtung:

Wir können $T_e = e;$ nur verschieben, falls e am Ende der Kante verfügbar ist:

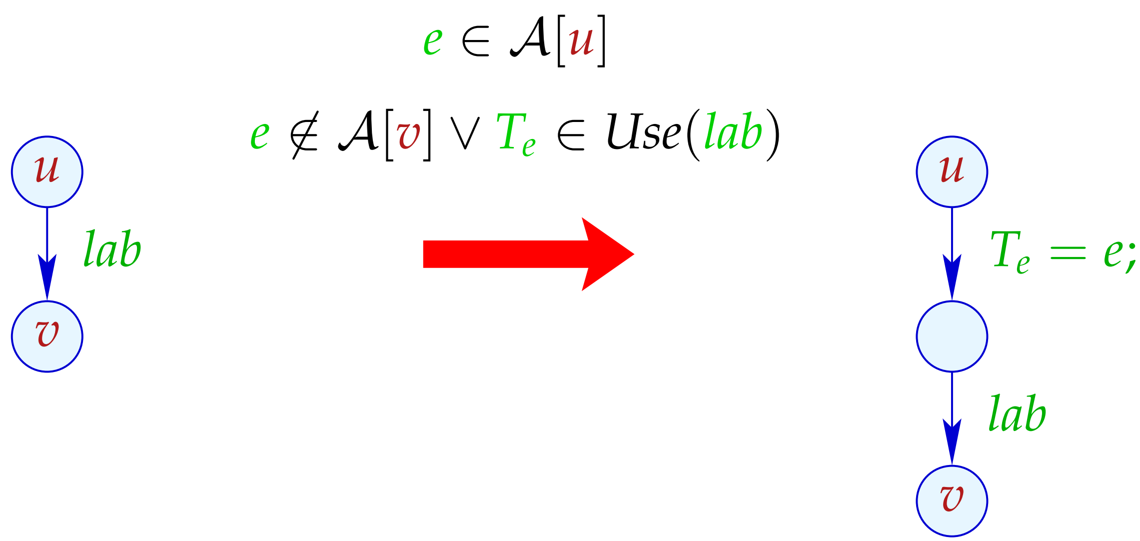


Offenbar ist $x + 1$ nicht an 1 verfügbar, darf also nicht verschoben werden !!!

Transformation 8.1:



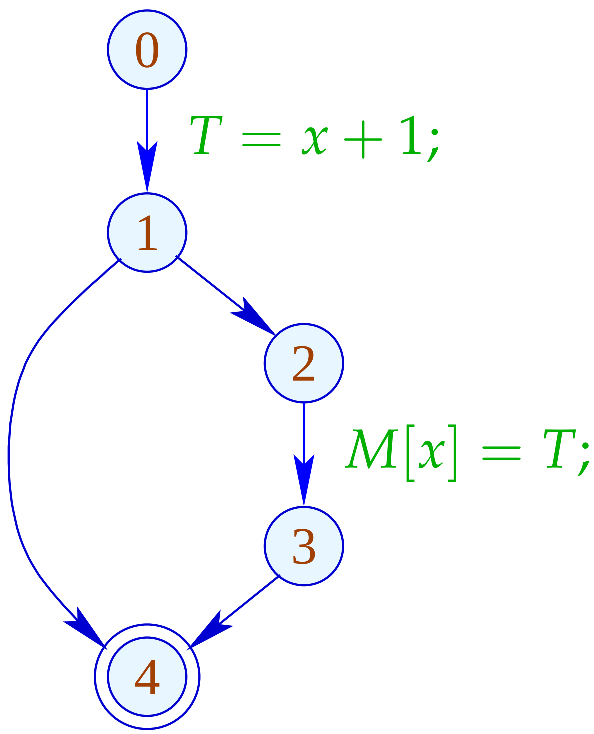
Transformation 8.2:



Beachte:

Die Transformation T8 ist nur sinnvoll, nachdem T1 gemacht wurde.

Im Beispiel beseitigt sie den partiell toten Code:

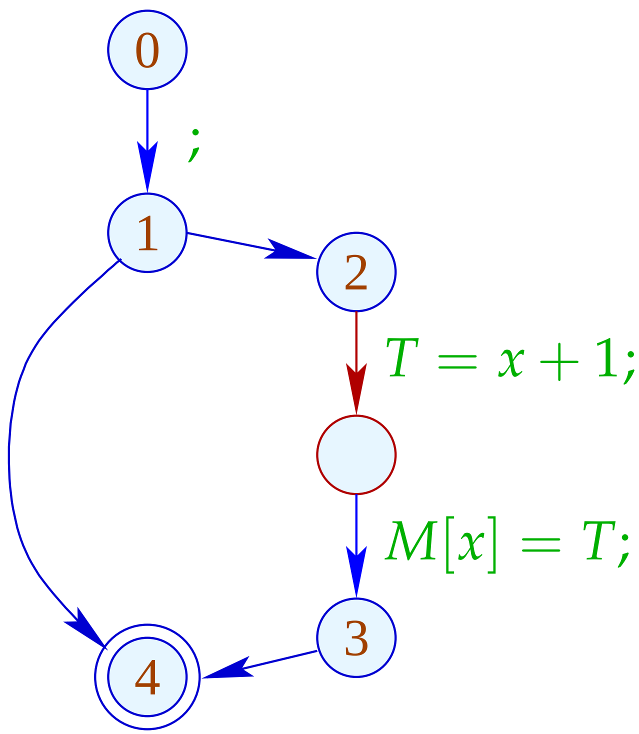


	$\mathcal{A}$
0	$\emptyset$
1	$\{x+1\}$
2	$\{x+1\}$
3	$\{x+1\}$
4	$\{x+1\}$

Beachte:

Die Transformation T8 ist nur sinnvoll, nachdem T1 gemacht wurde.

Im Beispiel beseitigt sie den partiell toten Code:



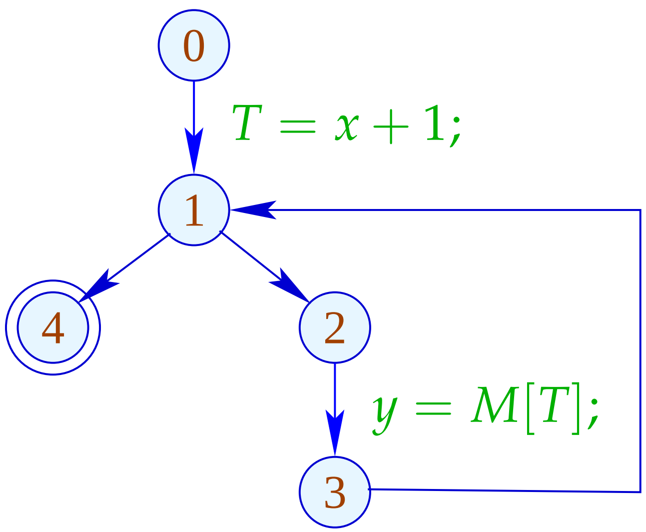
	$\mathcal{A}$
0	$\emptyset$
1	$\{x+1\}$
2	$\{x+1\}$
3	$\{x+1\}$
4	$\{x+1\}$

Fragen:

(1) Ist die Transformation korrekt ???

(2) Ist sie beweisbar nicht-verschlechternd ???

Beispiel:



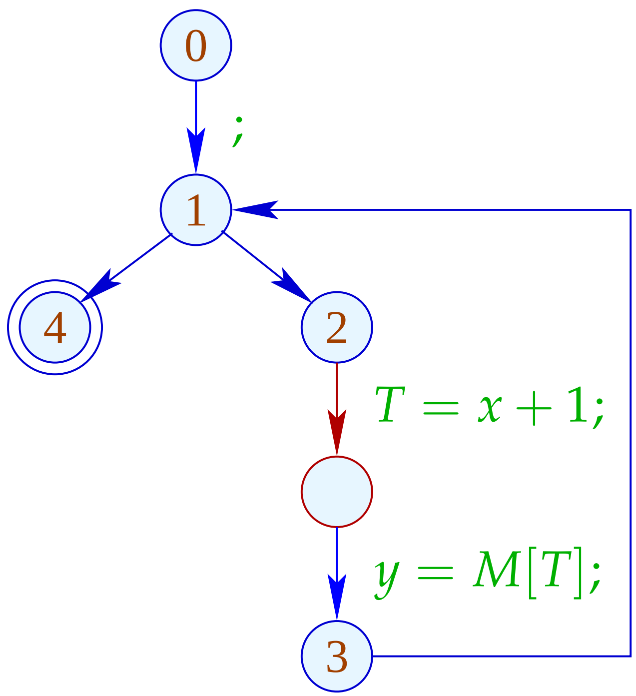
	$\mathcal{A}$
0	$\emptyset$
1	$\{x + 1\}$
2	$\{x + 1\}$
3	$\{x + 1\}$
4	$\{x + 1\}$

Fragen:

(1) Ist die Transformation korrekt ???

(2) Ist sie beweisbar nicht-verschlechternd ???

Beispiel:



	$\mathcal{A}$
0	$\emptyset$
1	$\{x + 1\}$
2	$\{x + 1\}$
3	$\{x + 1\}$
4	$\{x + 1\}$

Offenbar

... kann schleifen-invarianter Code entstehen ;-(

... kann Code dupliziert werden :-(

... kann eine Verschlechterung eintreten !!!

... kann diese Verschlechterung im Beispiel durch (Schleifen-Rotation und) PRE wieder beseitigt werden :-))

Immer?

Fazit:

- $\longrightarrow$ Das Design einer sinnvollen Optimierung ist nicht ganz einfach.
- $\longrightarrow$ Optimierungen, die ein Programm verbessern, können andere dramatisch verschlechtern :-(
- $\longrightarrow$ Manche Transformationen entfalten ihre Wirkung erst in Verbindung mit weiteren Optimierungen :-)
- $\longrightarrow$ Die Reihenfolge der angewandten Optimierungen ist entscheidend !!
- $\longrightarrow$ Manche Optimierungen können iteriert angewandt werden !!!

... eine sinnvolle Abfolge:

T5	Konstanten-Propagation Intervall-Analyse Alias-Analyse
T7	Schleifen-Rotation
T1	Hilfsvariablen für Ausdrücke
T2	verfügbare Ausdrücke
T4	Move-Optimierung
T3	tote Zuweisungen
T8	partiell toter Code
T6	partiell redundanter Code

2 Ersetzung teurer Berechnungen durch billigere

2.1 Reduction of Strength

(1) Polynom-Berechnung

$$f(x) = a_n \cdot x^n + a_{n-1} \cdot x^{n-1} + \ldots + a_1 \cdot x + a_0$$

	Multiplikationen	Additionen
naiv	$\frac{1}{2}n(n+1)$	n
Wiederverwendung	$2n-1$	n
Horner-Schema	n	n

Idee:

$$f(x) = (\ldots((a_n \cdot x + a_{n-1}) \cdot x + a_{n-2}) \ldots) \cdot x + a_0$$

(2) Tabellierung eines Polynoms $f(x)$ vom Grad n :

$\longrightarrow$ Für jeden Wert $f(x)$ neu auszuwerten ist zu teuer :-)

$\longrightarrow$ Glücklicherweise sind die n-ten Differenzen konstant !!!

Beispiel: $$f(x) = 3x^3 - 5x^2 + 4x + 13$$

n	$f(n)$	Δ	Δ^2	Δ^3
0	13	2	8	$\boxed{18}$
1	15	10	$\boxed{26}$	
2	25	$\boxed{36}$		
3	$\boxed{61}$			
4	$\ldots$			

Dabei ist die n-te Differenz stets

$$\Delta_h^n(f) = n! \cdot a_n \cdot h^n \qquad (h \text{ Schrittweite})$$

Kosten:

- n mal f auswerten;
- $\frac{1}{2} \cdot (n-1) \cdot n$ Subtraktionen, um die Δ^k zu berechnen;
- $2n-2$ Multiplikationen, um $\Delta_h^n(f)$ zu berechnen;
- n Additionen für jeden weiteren Wert :-)

$$\Longrightarrow$$

Anzahl der Multiplikationen hängt nur von n ab :-))

Einfacher Fall: $f(x) = a_1 \cdot x + a_0$

- ... kommt in vielen numerischen Schleifen vor :-)
- Die ersten Differenzen sind bereits konstant:

$$f(x+h) - f(x) = a_1 \cdot h$$

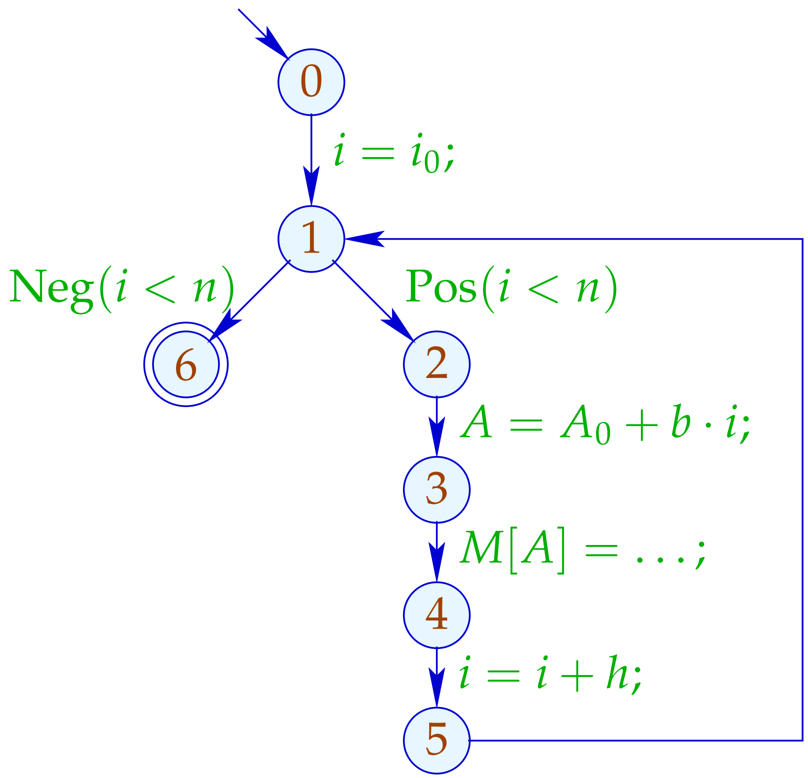
- Anstelle einer Folge: $y_i = f(x_0 + i \cdot h)\,,\quad i \geq 0$

  berechnen wir: $y_0 = f(x_0)\,,\quad \Delta = a_1 \cdot h$

  $y_i = y_{i-1} + \Delta\,,\quad i > 0$

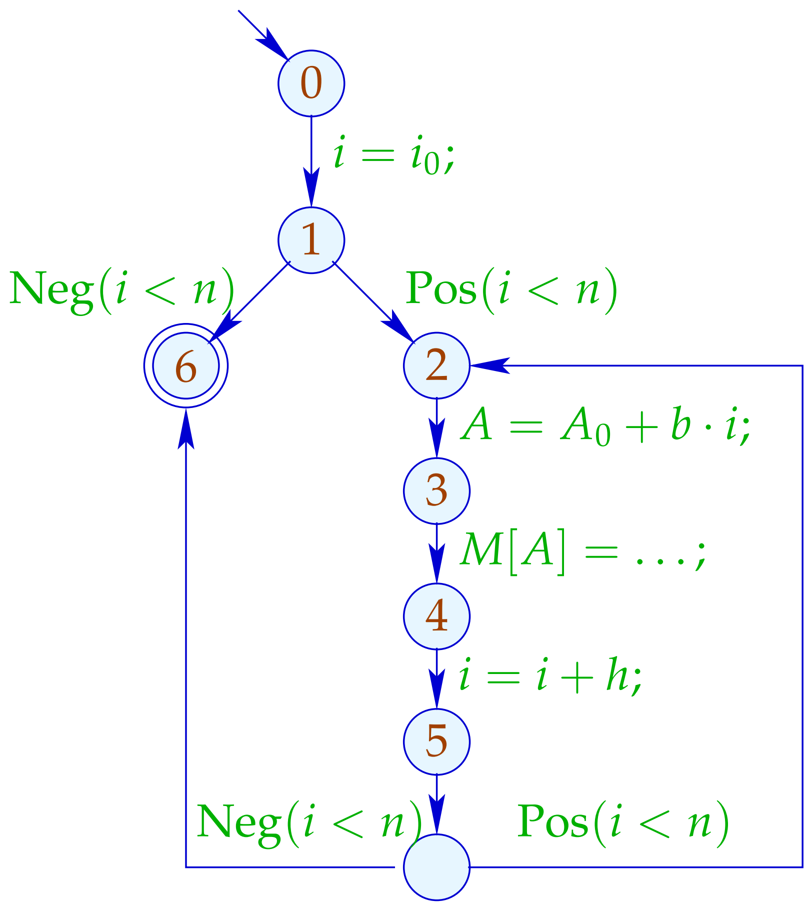
Beispiel:

$$\text{for } (i = i_0; i < n; i = i + h) \;\; \{$$
$$A = A_0 + b \cdot i;$$
$$M[A] = \ldots;$$
$$\}$$



... bzw. nach Schleifen-Rotation:

$$
\begin{array}{l}
i = i_0; \\
\textsf{if } (i < n) \textsf{ do } \{ \\
\qquad\qquad A = A_0 + b \cdot i; \\
\qquad\qquad M[A] = \ldots; \\
\qquad\qquad i = i + h; \\
\qquad \} \textsf{ while } (i < n);
\end{array}
$$



... und Reduktion der Stärke:

$i = i_0;$

if $(i < n)$ $\{$

$\quad \Delta = b \cdot h;$

$\quad A = A_0 + b \cdot i_0;$

$\quad$ do $\{$

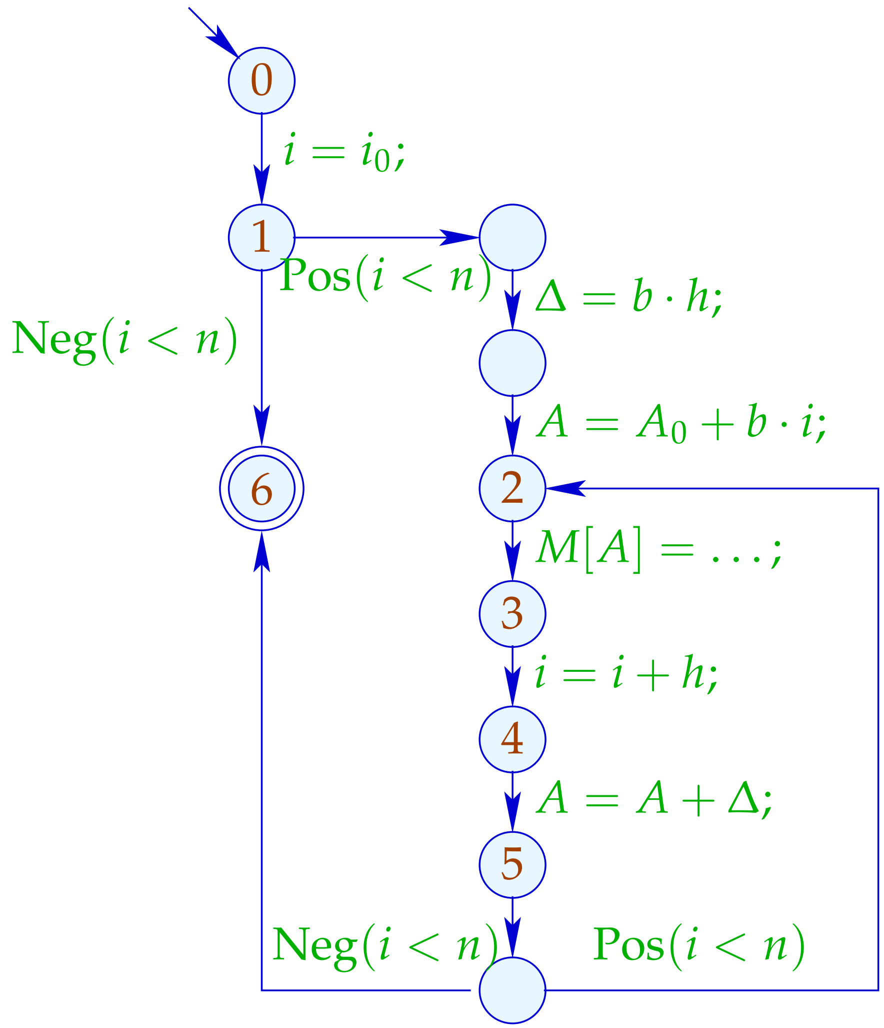
$\quad\quad M[A] = \ldots;$

$\quad\quad i = i + h;$

$\quad\quad A = A + \Delta;$

$\quad \}$ while $(i < n);$

$\}$



Achtung:

- Die Werte b, h, A_0 dürfen sich in der Schleife nicht ändern.
- i, A dürfen nur genau an einer Stelle in der Schleife modifiziert werden :-(
- Man könnte versuchen, die Variable i ganz einzusparen :

  $\longrightarrow$ i darf sonst nicht weiter benutzt werden.

  $\longrightarrow$ Man muss die Initialisierung transformieren in: $A = A_0 + b \cdot i_0$.

  $\longrightarrow$ Man muss die Schleifenbedingung $i < n$ transformieren in: $A < N$ für $N = A_0 + b \cdot n$.

  $\longrightarrow$ b muss ungleich Null sein !!!

Vorgehen:

Identifizieren von

... Schleifen;

... Iterations-Variablen;

... Konstanten;

... den richtigen Benutzungs-Strukturen.

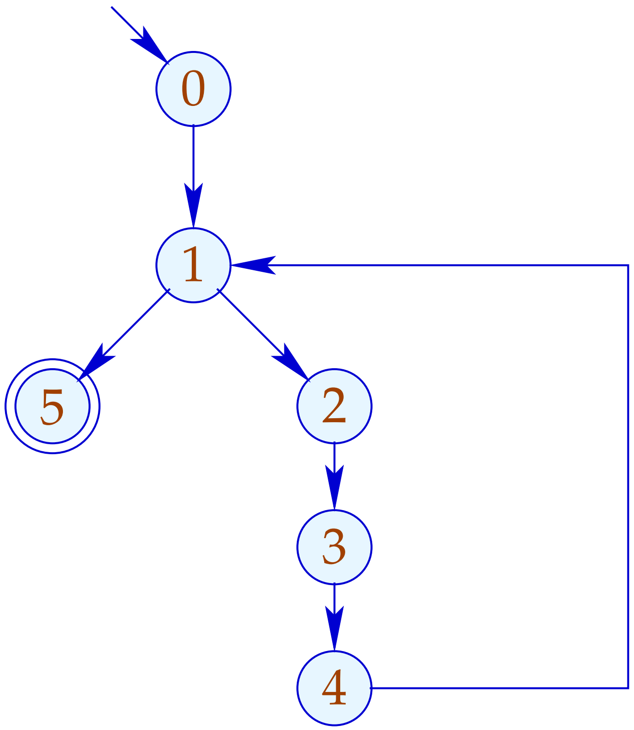
Schleifen:

... identifizieren wir durch einen Punkt v, zu dem ein Rücksprung $(\_, \_, v)$ existiert :-)

Für den Teilgraphen G_v des CFG auf $\{w \mid v \Rightarrow w\}$ definieren wir:

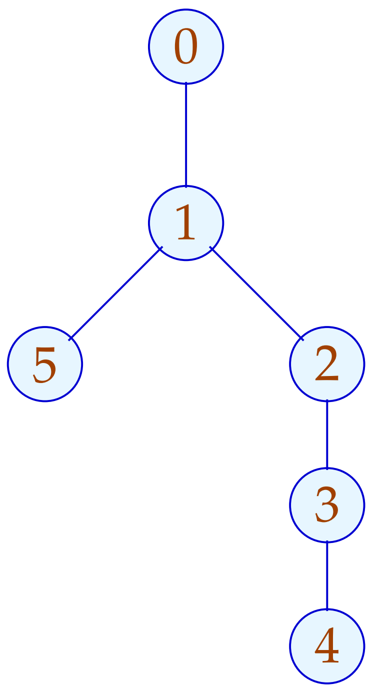
$$\mathsf{Loop}[v] \quad = \quad \{w \mid w \longrightarrow^* v \;\text{ in }\; G_v\}$$

Beispiel:



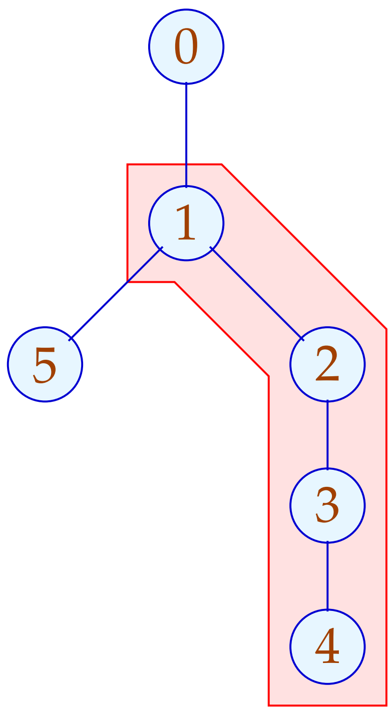
	$\mathcal{P}$
0	$\{0\}$
1	$\{0,1\}$
2	$\{0,1,2\}$
3	$\{0,1,2,3\}$
4	$\{0,1,2,3,4\}$
5	$\{0,1,5\}$

Beispiel:



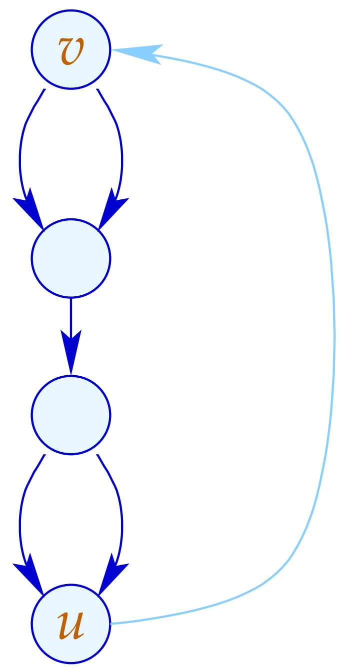
	$\mathcal{P}$
0	$\{0\}$
1	$\{0,1\}$
2	$\{0,1,2\}$
3	$\{0,1,2,3\}$
4	$\{0,1,2,3,4\}$
5	$\{0,1,5\}$

Beispiel:



	$\mathcal{P}$
0	$\{0\}$
1	$\{0,1\}$
2	$\{0,1,2\}$
3	$\{0,1,2,3\}$
4	$\{0,1,2,3,4\}$
5	$\{0,1,5\}$

Wir sind an Kanten interessiert, die pro Iteration exakt einmal ausgeführt werden:



Das ist graphentheoretisch nicht ganz leicht auszudrücken :-(

Man könnte solche Kanten k selektieren, dass:

- der Teilgraph $G = \mathsf{Loop}[v] \backslash \{(\_, \_, v)\}$ zusammenhängend ist;
- der Graph $G \backslash \{k\}$ in zwei unverbundene Teilgraphen zerfällt.

Man könnte solche Kanten k selektieren, dass:

- der Teilgraph $G = \mathsf{Loop}[v] \backslash \{(\_, \_, v)\}$ zusammenhängend ist;
- der Graph $G \backslash \{k\}$ in zwei unverbundene Teilgraphen zerfällt.

Auf der Source-Programm-Ebene ist das dagegen trivial:

$$\mathsf{do}\ \{\ s_1 \ldots s_k$$
$$\}\ \mathsf{while}\ (e);$$

Die gesuchten Zuweisungen müssen unter den s_i sein :-)

Iterationsvariable:

i heißt Iterationsvariable, wenn die einzige Definition von i in der Schleife an einer Kante erfolgt, die den Rumpf separiert, und von der Form:

$$i = i + h;$$

ist für eine Schleifen-Konstante h .

Eine Schleifen-Konstante ist einfach eine Konstante (z.B. 42) oder, etwas liberaler, ein Ausdruck, der nur von Variablen abhängt, die innerhalb der Schleife nicht modifiziert werden :-)

(3) Differenzen für Mengen

Betrachte die Fixpunkt-Berechnung:

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

Ist F distributiv, könnte man sie ersetzen durch:

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

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