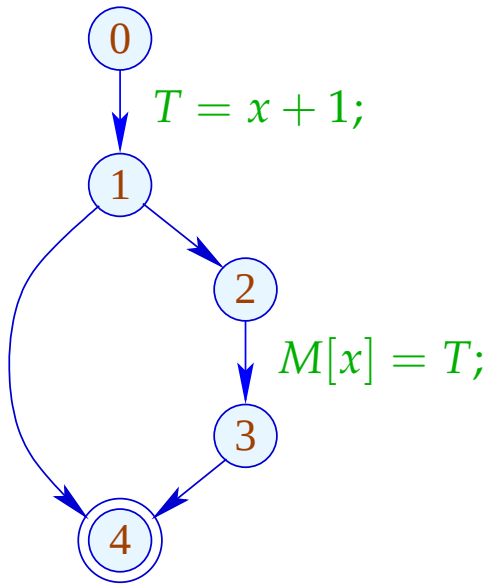
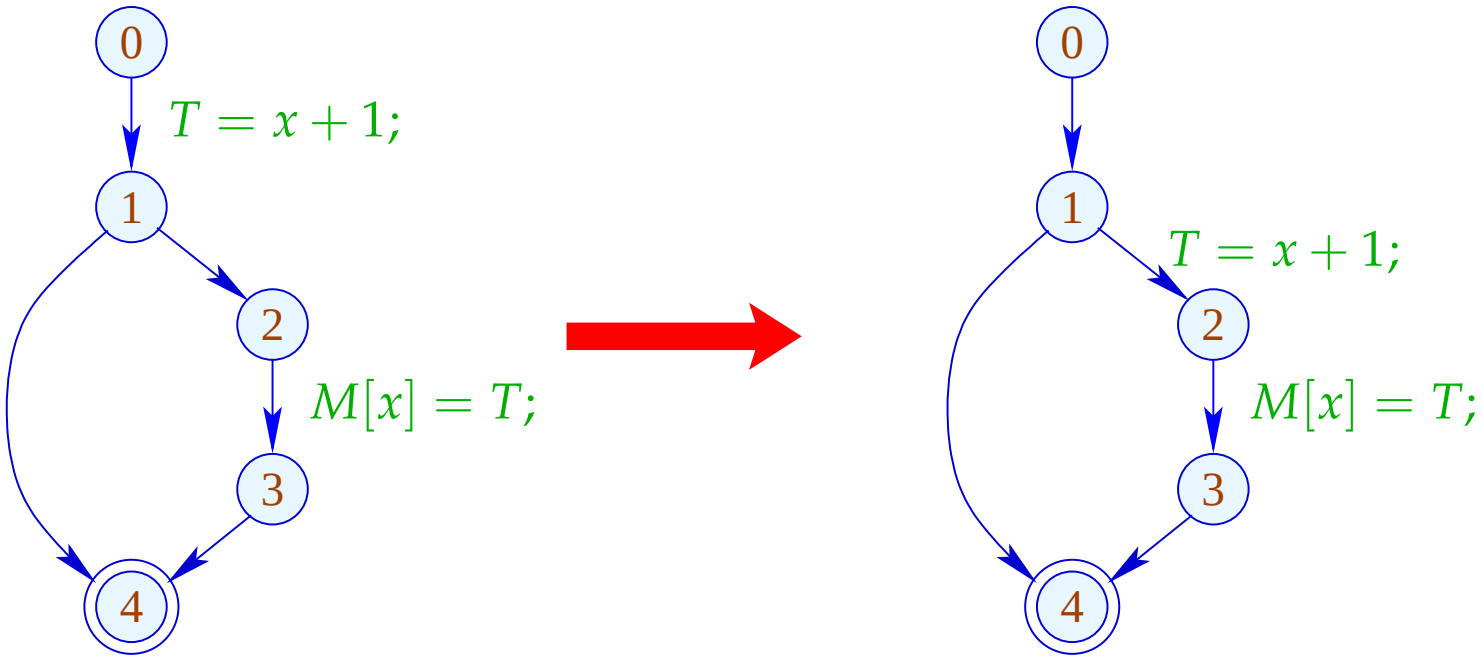
1.9 Beseitigung partiell toten Codes

Beispiel:



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

Idee:



Problem:

- Die Definition $x = e;$ $(x \notin Vars_e)$ darf nur dorthin geschoben werden, wo e sicher ist ;-)
- Die Definition muss weiterhin für die Benutzungen von x zur Verfügung stehen ;-)

$\Longrightarrow$

Wir definieren eine neue Analyse, welche Berechnungen maximal verzögert.

$$[\![;]\!]^\sharp D = $$

$$[\![x = e;]\!]^\sharp D = \begin{cases} D \backslash (Use_e \cup Def_x) \cup \{x = e;\} & \text{falls} \quad x \notin Vars_e \\ D \backslash (Use_e \cup Def_x) & \text{falls} \quad x \in Vars_e \end{cases}$$

Dabei ist:

$$
\begin{aligned}
Use_e &= \{y = e'; \mid y \in Vars_e\} \\
Def_x &= \{y = e'; \mid y \equiv x \vee x \in Vars_{e'}\}
\end{aligned}
$$

Dabei ist:

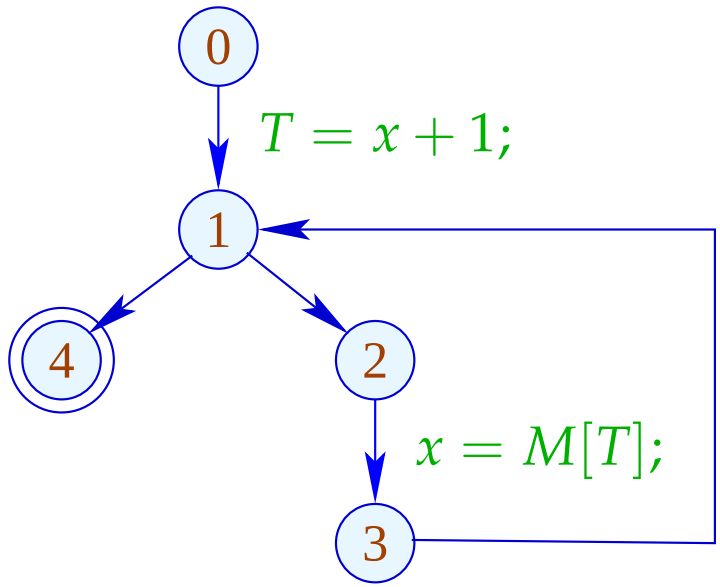
$$\begin{aligned} Use_e &= \{y = e';\mid y \in Vars_e\} \\ Def_x &= \{y = e';\mid y \equiv x \vee x \in Vars_{e'}\} \end{aligned}$$

Für die anderen Kanten definieren wir:

$$\begin{aligned} [\![x = M[e];]\!]^\sharp D &= D \backslash (Use_e \cup Def_x) \\ [\![M[e_1] = e_2;]\!]^\sharp D &= D \backslash (Use_{e_1} \cup Use_{e_2}) \\ [\![\mathsf{Pos}(e)]\!]^\sharp D &= [\![\mathsf{Neg}(e)]\!]^\sharp D = D \backslash Use_e \end{aligned}$$

Achtung:

Wir können $y = e;$ nur über einen Zusammenfluss von Kanten verschieben, falls $y = e;$ entlang aller Kanten verschoben werden kann:



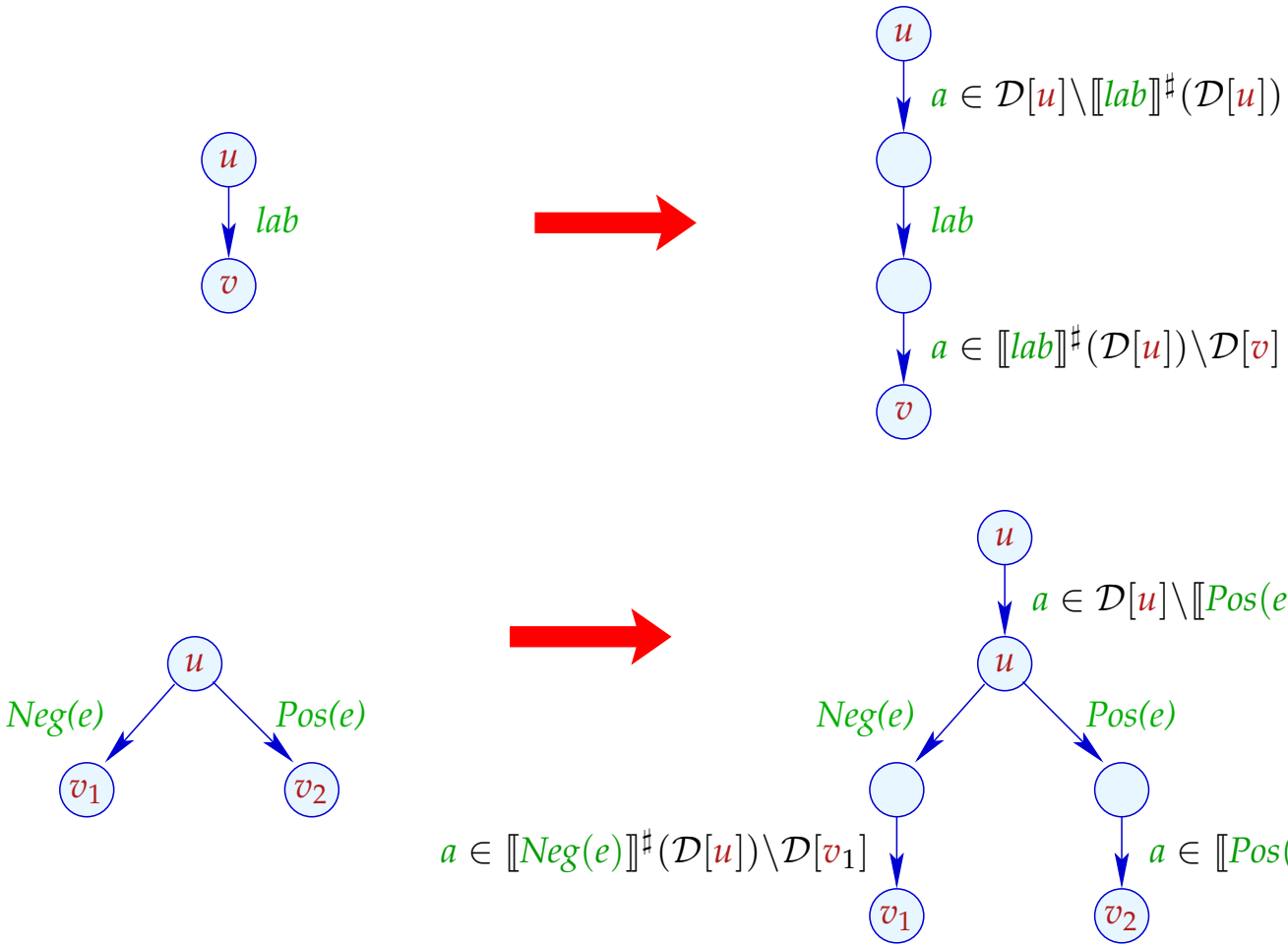
Offenbar kann $T = x + 1;$ nicht über 1 hinaus verschoben werden !!!

Wir schließen:

- Die Ordnung auf dem Verband für Verzögerbarkeit ist "$\supseteq$".
- Am Anfang des Programms gilt: $D_0 = \emptyset$.

  Damit können wir die Mengen $\mathcal{D}[u]$ der an u durch Verzögerung ankommenden Zuweisungen durch Lösen eines Ungleichungssystems berechnen.
- Wir verzögern nur Zuweisungen a bei denen $a\,a$ den gleichen Effekt hat wie a allein.
- Durch weiteres Einfügen werden die Zuweisungen an der ursprünglichen Position Zuweisungen an tote Variablen ...

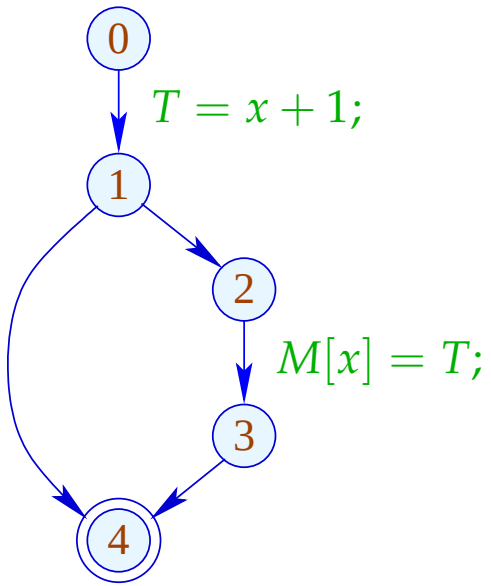
Transformation 7:



Beachte:

Die Transformation T7 ist nur sinnvoll, wenn wir anschließend mit T2 Zuweisungen an tote Variablen beseitigen :-)

Im Beispiel beseitigt sie den partiell toten Code:

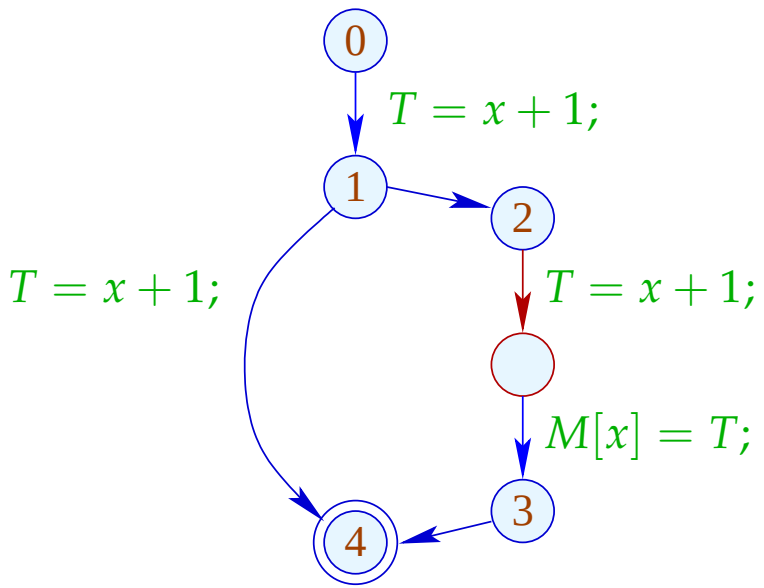


	$\mathcal{D}$
0	$\emptyset$
1	$\{T = x + 1;\}$
2	$\{T = x + 1;\}$
3	$\emptyset$
4	$\emptyset$

Beachte:

Die Transformation T7 ist nur sinnvoll, wenn wir anschließend mit T2 Zuweisungen an tote Variablen beseitigen :-)

Im Beispiel beseitigt sie den partiell toten Code:

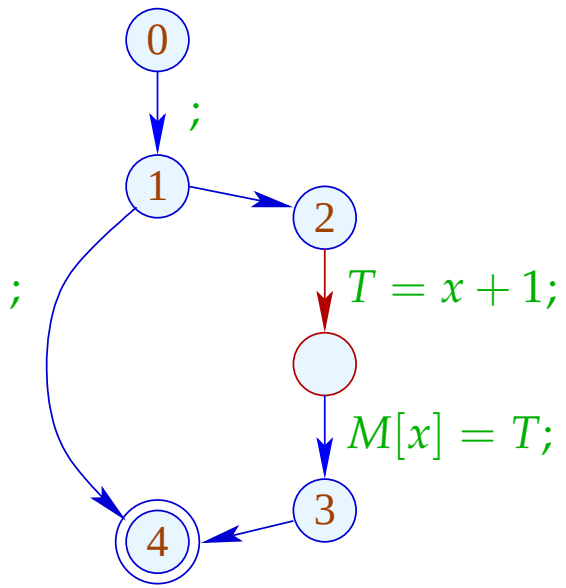


	$\mathcal{D}$
0	$\emptyset$
1	$\{T = x + 1;\}$
2	$\{T = x + 1;\}$
3	$\emptyset$
4	$\emptyset$

Beachte:

Die Transformation T7 ist nur sinnvoll, wenn wir anschließend mit T2 Zuweisungen an tote Variablen beseitigen :-)

Im Beispiel beseitigt sie den partiell toten Code:



	$\mathcal{L}$
0	$\{x\}$
1	$\{x\}$
2	$\{x\}$
$2'$	$\{x, T\}$
3	$\emptyset$
4	$\emptyset$

Bemerkungen:

- Nach $T7$ sind sämtliche ursprünglichen Zuweisungen $y = e;$ mit $y \notin Vars_e$ Zuweisungen an tote Variablen und können deshalb stets gestrichen werden :-)
- Damit kann man erneut zeigen, dass die Transformation garantiert nicht verschlechternd ist :-))
- Wie bei der Beseitigung partieller Redundanzen kann die Transformation mehrmals ausgeführt werden :-}

Fazit:

- → Das Design einer sinnvollen Optimierung ist nicht ganz einfach.
- → Manche Transformationen entfalten ihre Wirkung erst in Verbindung mit weiteren Optimierungen :-)
- → Die Reihenfolge der angewandten Optimierungen ist entscheidend !!
- → Manche Optimierungen können iteriert angewandt werden !!!

... eine sinnvolle Abfolge:

T4	Konstanten-Propagation Intervall-Analyse Alias-Analyse
T6	Schleifen-Rotation
T1, T3, T2	verfügbare Ausdrücke
T2	tote Zuweisungen
T7, T2	partiell toter Code
T5, T3, T2	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 :

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

$\rightarrow$ 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	18
1	15	10	26	
2	25	36		
3	61			
4	...			

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

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

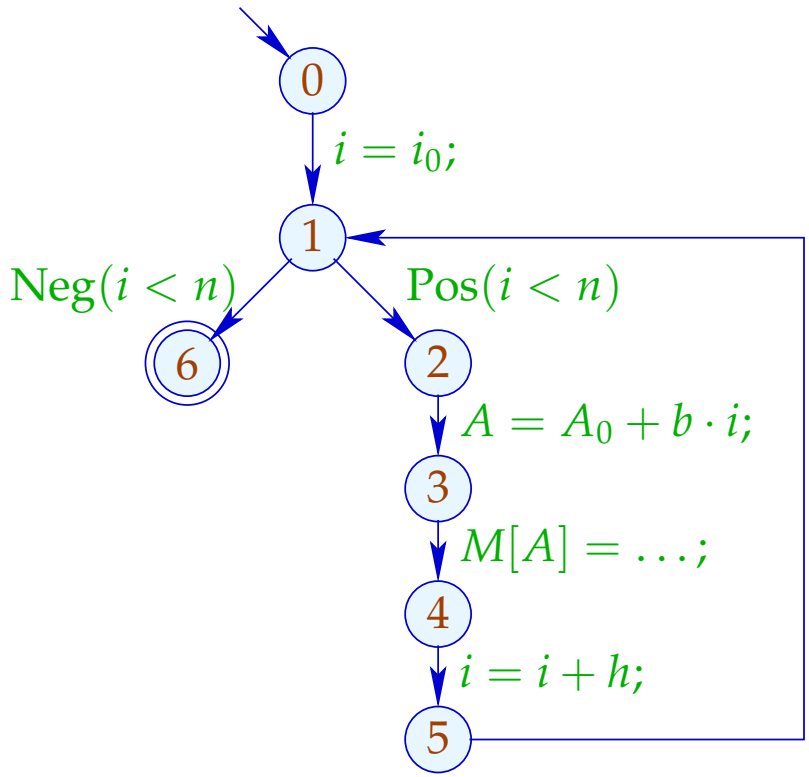
Beispiel:

for $(i = i_0; i < n; i = i + h)$ {

$$A = A_0 + b \cdot i;$$

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

}



... bzw. nach Schleifen-Rotation:

$i = i_0;$

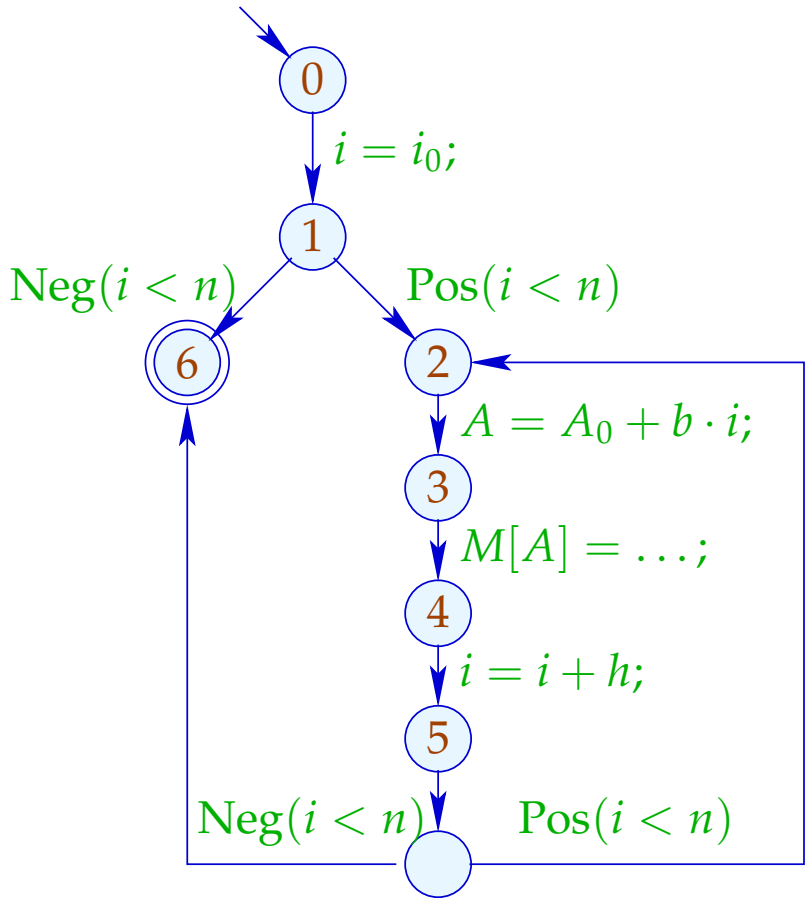
if $(i < n)$ do $\{$

$A = A_0 + b \cdot i;$

$M[A] = \ldots;$

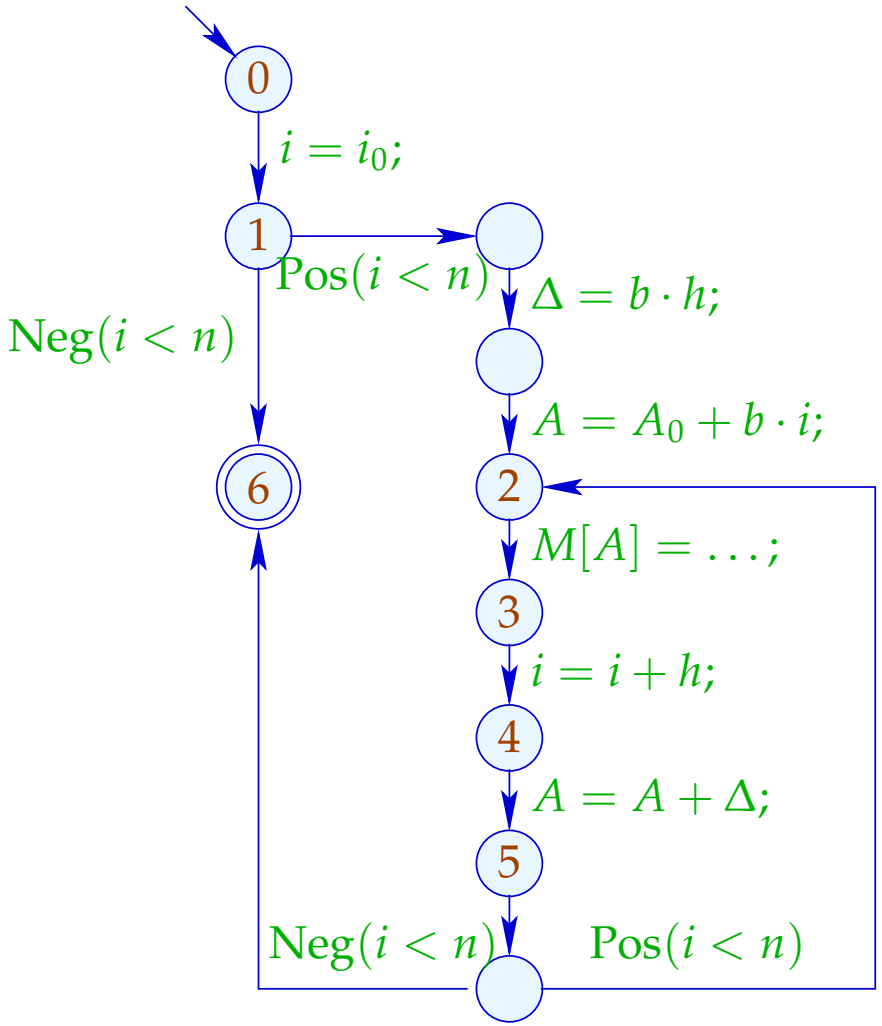
$i = i + h;$

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



... und Reduktion der Stärke:

$i = i_0;$
if $(i < n)$ {
  $\Delta = b \cdot h;$
  $A = A_0 + b \cdot i_0;$
  do {
    $M[A] = \ldots;$
    $i = i + h;$
    $A = A + \Delta;$
  } 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 :

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

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

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

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

Vorgehen:

Identifizieren von

- . . . Schleifen;
- . . . Iterations-Variablen;
- . . . Konstanten;
- . . . den richtigen Benutzungs-Strukturen.

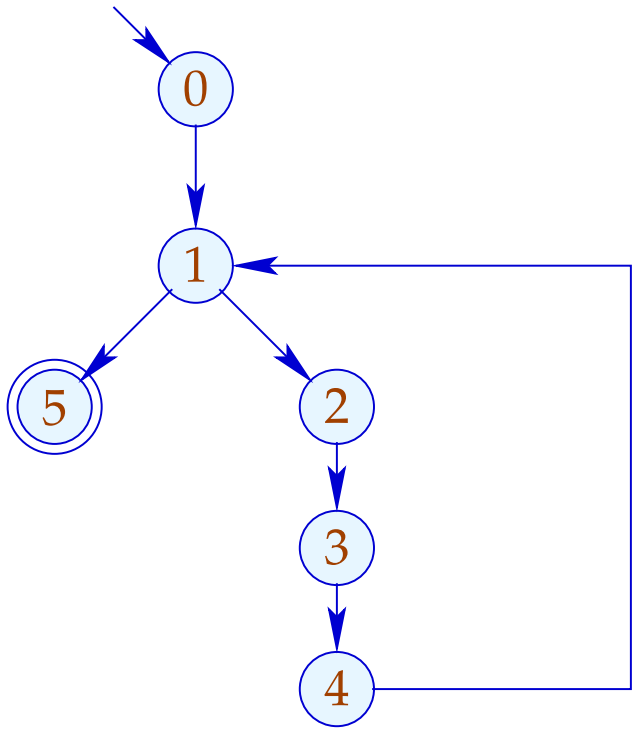
Schleifen:

... 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:

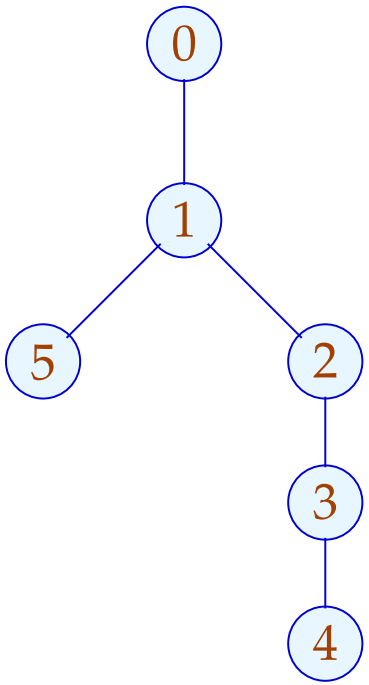
$$\mathsf{Loop}[v] \quad = \quad \{w \mid w \rightarrow^* v \ \text{ in } \ G_v\}$$

Beispiel:



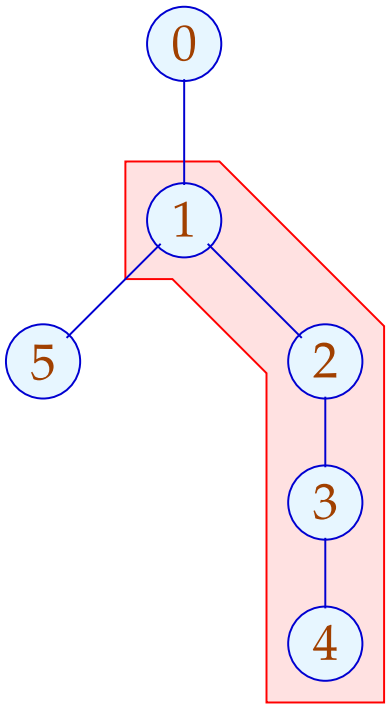
	$\mathcal{P}$
0	$\{0\}$
1	$\{0,1\}$
2	$\{0,1,2\}$
3	$\{0,1,2,3\}$
4	$\{0,1,2,3,4\}$
5	$\{0,1,5\}$

Beispiel:



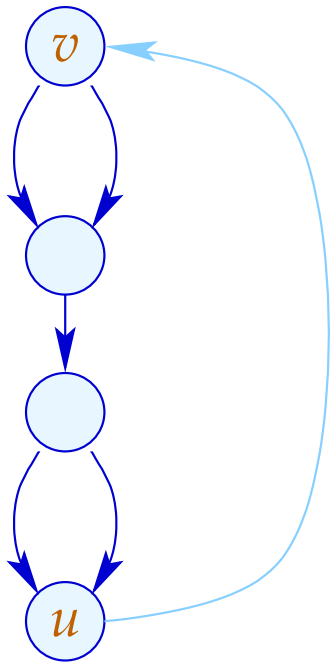
	$\mathcal{P}$
0	$\{0\}$
1	$\{0,1\}$
2	$\{0,1,2\}$
3	$\{0,1,2,3\}$
4	$\{0,1,2,3,4\}$
5	$\{0,1,5\}$

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:

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

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

Statt der Folge: $\emptyset \subseteq F(\emptyset) \subseteq F^2(\emptyset) \subseteq \ldots$

berechnen wir: $\Delta_1 \cup \Delta_2 \cup \ldots$

wobei: $\Delta_{i+1} = F(F^i(\emptyset)) \setminus F^i(\emptyset)$

$= F(\Delta_i) \setminus (\Delta_1 \cup \ldots \cup \Delta_i)$ mit $\Delta_0 = \emptyset$

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

Dann summieren sich die Kosten zu:

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

wobei n die Kardinalität des Ergebnisses ist.

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