Helmut Seidl

Programmoptimierung

TU München

Wintersemester 2004/05

Organisatorisches

Termine:

Vorlesung:	Montag, 12-14
	Dienstag, 12-14
Übung:	Freitag, 10-12
	K. Neeraj Verma: `verma@in.tum.de`
Materialien:	Folien, Aufzeichnung :-)
	Literatur :-))
	Vorlesungs-Mitschrift (in Überarbeitung)

Schein:

- 50% der Aufgaben
- zweimal vorrechnen :-)

Geplanter Inhalt:

1. Vermeidung überflüssiger Berechnungen

   $\longrightarrow$ verfügbare Ausdrücke

   $\longrightarrow$ Konstantenpropagation/Array-Bound-Checks

   $\longrightarrow$ Code Motion

2. Ersetzen teurer Berechnungen durch billige

   $\longrightarrow$ Peep Hole Optimierung

   $\longrightarrow$ Inlining

   $\longrightarrow$ Reduction of Strength

   ...

3. Anpassung an Hardware

$\longrightarrow$ Instruktions-Selektion

$\longrightarrow$ Registerverteilung

$\longrightarrow$ Scheduling

$\longrightarrow$ Speicherverwaltung

0 Einführung

Beobachtung 1: Intuitive Programme sind oft ineffizient.

Beispiel:

```
void swap (int i, int j) {
    int t;
    if (a[i] > a[j]) {
        t = a[j];
        a[j] = a[i];
        a[i] = t;
    }
}
```

Ineffizienzen:

- Adressen `a[i]`, `a[j]` werden je dreimal berechnet :-(
- Werte `a[i]`, `a[j]` werden zweimal geladen :-(

Verbesserung:

- Gehe mit Pointer durch das Feld `a`;
- speichere die Werte von `a[i]`, `a[j]` zwischen!

```
void swap (int *p, int *q) {
     int t, ai, aj;
     ai = *p; aj = *q;
     if (ai > aj) {
         t = aj;
         *q = ai;
         *p = t;     // t kann auch noch
        }            // eingespart werden!
     }
```

Beobachtung 2:

Höhere Programmiersprachen (sogar C :-) abstrahieren von Hardware und Effizienz.

Aufgabe des Compilers ist es, den natürlich erzeugten Code an die Hardware anzupassen.

Beispiele:

... Füllen von Delay-Slots;

... Einsatz von Spezialinstruktionen;

... Umorganisation der Speicherzugriffe für besseres Cache-Verhalten;

... Beseitigung (unnötiger) Tests auf Overflow/Range.

Beobachtung 3:

Programm-Verbesserungen sind nicht immer korrekt :-(

Beispiel:

`y = f() + f();` $\Longrightarrow$ `y = 2 * f();`

Idee:

Spare zweite Auswertung von `f()` ...

Beobachtung 3:

Programm-Verbesserungen sind nicht immer korrekt :-(

Beispiel:

```
y = f() + f();
```

$\Longrightarrow$

```
y = 2 * f();
```

Idee: Spare zweite Auswertung von `f()` ???

Problem: Die zweite Auswertung könnte ein anderes Ergebnis liefern als die erste (z.B. wenn `f()` aus der Eingabe liest :-)

Folgerungen:

$\Longrightarrow$ Optimierungen haben Voraussetzungen.

$\Longrightarrow$ Die Voraussetzungen muss man:

- formalisieren,
- überprüfen :-)

$\Longrightarrow$ Man muss beweisen, dass die Optimierung korrekt ist, d.h. die Semantik erhält !!!

Beobachtung 4:

Optimierungs-Techniken hängen von der Programmiersprache ab:

- $\rightarrow$ welche Ineffizienzen auftreten;
- $\rightarrow$ wie gut sich Programme analysieren lassen;
- $\rightarrow$ wie schwierig / unmöglich es ist, Korrektheit zu beweisen ...

Beispiel: Java

Unvermeidbare Ineffizienzen:

- $*$ Array-Bound Checks;
- $*$ dynamische Methoden-Auswahl;
- $*$ bombastische Objekt-Organisation ...

Analysierbarkeit:

- $+$ keine Pointer-Arithmetik;
- $+$ keine Pointer in den Stack;
- $-$ dynamisches Klassenladen;
- $-$ Reflection, Exceptions, Threads, ...

Korrektheitsbeweise:

- $+$ mehr oder weniger definierte Semantik;
- $-$ Features, Features, Features;
- $-$ Bibliotheken mit wechselndem Verhalten ...

... in der Vorlesung:

eine einfache imperative Sprache mit:

• Variablen	//	Register
• $R = e;$	//	Zuweisungen
• $R_1 = M[R_2];$	//	Laden
• $M[R_1] = R_2;$	//	Speichern
• if (e) s_1 else s_2	//	bedingte Verzweigung
• goto $L;$	//	keine Schleifen :-)

Beachte:

- Vorerst verzichten wir auf Prozeduren :-)
- Externe Funktionen berücksichtigen wir, indem wir als Ausdruck e auch $f(R_1, \ldots, R_k)$ gestatten für eine unbekannte Funktion f.

  $\Longrightarrow$ intra-prozedural

  $\Longrightarrow$ eine Art Zwischensprache, in die man (fast) alles übersetzen kann.

Beispiel: `swap()`

$$
\begin{array}{llll}
A_1 & = & A_0 + 1 * i; & // \quad A_0 == \&a \\
R_1 & = & M[A_1]; & // \quad R_1 == a[i] \\
A_2 & = & A_0 + 1 * j; & \\
R_2 & = & M[A_2]; & // \quad R_2 == a[j] \\
\multicolumn{4}{l}{\text{if } (R_1 > R_2) \ \{} \\
\quad A_3 & = & A_0 + 1 * j; & \\
\quad t & = & M[A_3]; & \\
\quad A_4 & = & A_0 + 1 * j; & \\
\quad A_5 & = & A_0 + 1 * i; & \\
\quad R_3 & = & M[A_5]; & \\
\quad M[A_4] & = & R_3; & \\
\quad A_6 & = & A_0 + 1 * i; & \\
\quad M[A_6] & = & t; & \\
\quad \} & & &
\end{array}
$$

Optimierung 1: $1 * R \implies R$

Optimierung 2: Wiederbenutzung von Teilausdrücken

$$A_1 == A_5 == A_6$$
$$A_2 == A_3 == A_4$$

$$M[A_1] == M[A_5]$$
$$M[A_2] == M[A_3]$$

$$R_1 == R_3$$

Damit erhalten wir:

$$
\begin{array}{lcl}
A_1 & = & A_0 + i; \\
R_1 & = & M[A_1]; \\
A_2 & = & A_0 + j; \\
R_2 & = & M[A_2]; \\
\multicolumn{3}{l}{\textsf{if } (R_1 > R_2) \; \{} \\
\end{array}
$$

$$
\begin{array}{lcl}
t & = & R_2; \\
M[A_2] & = & R_1; \\
M[A_1] & = & t; \\
\}
\end{array}
$$

Optimierung 3: Verkürzung von Zuweisungsketten :-)

Ersparnis:

	vorher	nachher
$+$	6	2
$*$	6	0
load	4	2
store	2	2
$>$	1	1
$=$	6	2

1 Vermeidung überflüssiger Berechnungen

1.1 Mehrfach-Berechnungen

Idee:

Wird der gleiche Wert mehrfach berechnet, dann

$\rightarrow$ speichere ihn nach der ersten Berechnung;

$\rightarrow$ ersetze jede weitere Berechnung durch Nachschlagen!

$\Longrightarrow$ Verfügbarkeit von Ausdrücken

$\Longrightarrow$ Memoisierung

Problem: Erkenne Mehrfach-Berechnungen!

Beispiel:

$$\begin{array}{lrcl} & z & = & 1; \\ & y & = & \mathsf{read}(); \\ A: & x_1 & = & \boxed{y+z}; \\ & & \dots & \\ B: & x_2 & = & \boxed{y+z}; \end{array}$$

Achtung:

B ist eine Mehrfach-Berechnung des Werts von $\boxed{y+z}$, falls:

(1) A stets vor B ausgeführt wird; und

(2) y und z an B die gleichen Werte haben wie an A :-)

$\Longrightarrow$ Wir benötigen

$\longrightarrow$ eine operationelle Semantik :-)

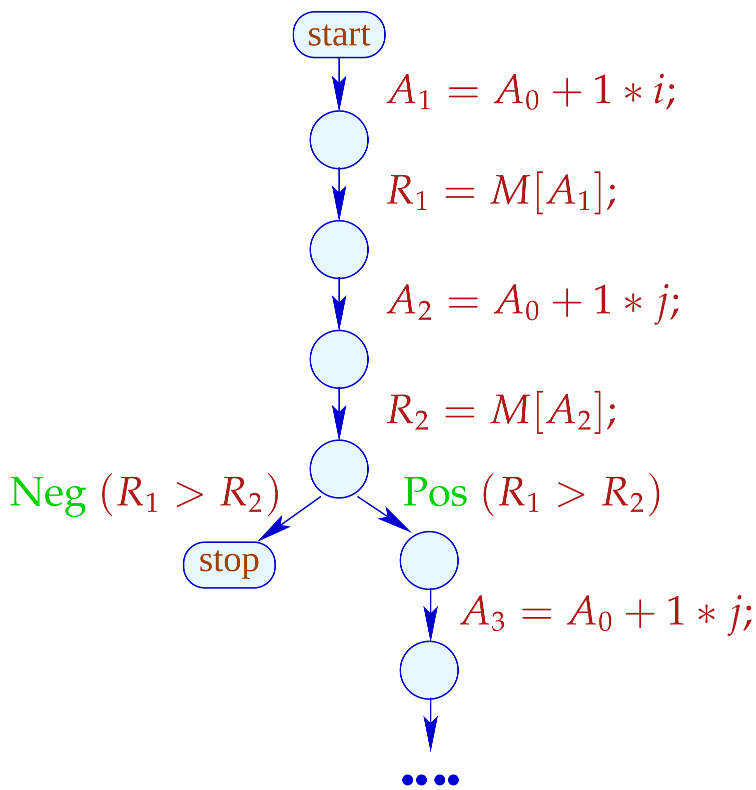
$\longrightarrow$ ein Verfahren, das einige Mehrfach-Berechnungen erkennt ...

Exkurs 1: Eine operationelle Semantik

Wir wählen einen small-step operationellen Ansatz.

Programme repräsentieren wir als Kontrollfluss-Graphen.

Im Beispiel:



Dabei repräsentieren:

Knoten	Programm-Punkt
start	Programm-Anfang
stop	Programm-Ende
Kante	Berechnungs-Schritt

Dabei repräsentieren:

Knoten	Programm-Punkt
start	Programm-Anfang
stop	Programm-Ende
Kante	Berechnungs-Schritt

Kanten-Beschriftungen:

Test : Pos (e) oder Neg (e)

Zuweisung : $R = e;$

Load : $R_1 = M[R_2];$

Store : $M[R_1] = R_2;$

Nop : $;$

Berechnungen folgen Pfaden.

Berechnungen transformieren den aktuellen Zustand

$$s = (\rho, \mu)$$

wobei:

$\rho : \mathit{Vars} \to \mathbf{int}$	Inhalt der Register
$\mu : \mathbb{N} \to \mathbf{int}$	Inhalt des Speichers

Jede Kante $k = (u, \mathit{lab}, v)$ definiert eine partielle Transformation

$$[\![k]\!] = [\![\mathit{lab}]\!]$$

des Zustands:

$$
\begin{array}{lcll}
[\![;]\!]\,(\rho,\mu) & = & (\rho,\mu) & \\
 & & & \\
[\![\mathrm{Pos}\,(e)]\!]\,(\rho,\mu) & = & (\rho,\mu) & \text{falls } [\![e]\!]\,\rho \neq 0 \\
[\![\mathrm{Neg}\,(e)]\!]\,(\rho,\mu) & = & (\rho,\mu) & \text{falls } [\![e]\!]\,\rho = 0
\end{array}
$$

$$[\![;]\!]\,(\rho, \mu) \quad = \quad (\rho, \mu)$$

$$[\![\mathrm{Pos}\,(e)]\!]\,(\rho, \mu) \quad = \quad (\rho, \mu) \qquad \text{falls } [\![e]\!]\,\rho \neq 0$$

$$[\![\mathrm{Neg}\,(e)]\!]\,(\rho, \mu) \quad = \quad (\rho, \mu) \qquad \text{falls } [\![e]\!]\,\rho = 0$$

// $[\![e]\!]$: Auswertung des Ausdrucks e, z.B.

// $[\![x + y]\!]\,\{x \mapsto 7, y \mapsto -1\} = 6$

// $[\![!(x == 4)]\!]\,\{x \mapsto 5\} = 1$

$$[\![;]\!]\,(\rho, \mu) \quad = \quad (\rho, \mu)$$

$$[\![\text{Pos}\,(e)]\!]\,(\rho, \mu) \quad = \quad (\rho, \mu) \qquad \text{falls } [\![e]\!]\,\rho \neq 0$$
$$[\![\text{Neg}\,(e)]\!]\,(\rho, \mu) \quad = \quad (\rho, \mu) \qquad \text{falls } [\![e]\!]\,\rho = 0$$

// $[\![e]\!]$: Auswertung des Ausdrucks e, z.B.

// $[\![x + y]\!]\,\{x \mapsto 7, y \mapsto -1\} = 6$

// $[\![!(x == 4)]\!]\,\{x \mapsto 5\} = 1$

$$[\![R = e;]\!]\,(\rho, \mu) \quad = \quad (\boxed{\rho \oplus \{R \mapsto [\![e]\!]\,\rho\}}, \mu)$$

// wobei "$\oplus$" eine Abbildung an einer Stelle ändert

$$[\![R_1 = M[R_2];]\!]\,(\rho,\mu) \quad = \quad (\boxed{\rho \oplus \{R_1 \mapsto \mu(\rho(R_2))\}}, \mu)$$

$$[\![M[R_1] = R_2;]\!]\,(\rho,\mu) \quad = \quad (\rho, \boxed{\mu \oplus \{\rho(R_1) \mapsto \rho(R_2)\}})$$

Beispiel:

$[\![x = x + 1;]\!]\,(\{x \mapsto 5\}, \mu) = (\rho, \mu)$ wobei:

$$\begin{array}{rcl} \rho & = & \{x \mapsto 5\} \oplus \{x \mapsto [\![x+1]\!]\,\{x \mapsto 5\}\} \\ & = & \{x \mapsto 5\} \oplus \{x \mapsto 6\} \\ & = & \{x \mapsto 6\} \end{array}$$

Ein Pfad $\pi = k_1 k_2 \ldots k_m$ ist eine Berechnung für den Zustand s falls:

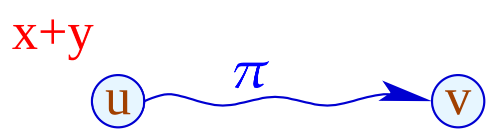
$$s \in def \left([\![k_m]\!] \circ \ldots \circ [\![k_1]\!] \right)$$

Das Ergebnis der Berechnung ist:

$$[\![\pi]\!]\, s = \left([\![k_m]\!] \circ \ldots \circ [\![k_1]\!] \right) s$$

Anwendung:

Nehmen wir an, wir hätten am Punkt u den Wert von $x + y$ berechnet:



Wir führen eine Berechnung entlang des Pfades π aus und erreichen v, wo wir erneut $x + y$ berechnen sollen ...

Idee:

Wenn x und y in π nicht verändert werden, dann muss $x + y$ in v den gleichen Wert liefern wie in u :-)

Diese Eigenschaft können wir an jeder Kante in π überprüfen :-}

Idee:

Wenn x und y in π nicht verändert werden, dann muss $x + y$ in v den gleichen Wert liefern wie in u :-)

Diese Eigenschaft können wir an jeder Kante in π überprüfen :-}

Allgemeiner:

Nehmen wir an, in u hätten wir die Werte der Ausdrücke aus $A = \{e_1, \ldots, e_r\}$ zur Verfügung.

Idee:

Wenn x und y in π nicht verändert werden, dann muss $x + y$ in v den gleichen Wert liefern wie in u :-)

Diese Eigenschaft können wir an jeder Kante in π überprüfen :-}

Allgemeiner:

Nehmen wir an, in u hätten wir die Werte der Ausdrücke aus $A = \{e_1, \ldots, e_r\}$ zur Verfügung.

Jede Kante k transformiert diese Menge in eine Menge $[\![k]\!]^\sharp\, A$ von Ausdrücken, die nach Ausführung von k verfügbar sind ...

... die wir zur Ermittlung des Effekts eines Pfads $\pi = k_1 \ldots k_r$ zusammen setzen können:

$$[\![\pi]\!]^\sharp = [\![k_r]\!]^\sharp \circ \ldots \circ [\![k_1]\!]^\sharp$$

... die wir zur Ermittlung des Effekts eines Pfads $\pi = k_1 \ldots k_r$ zusammen setzen können:

$$[\![\pi]\!]^\sharp = [\![k_r]\!]^\sharp \circ \ldots \circ [\![k_1]\!]^\sharp$$

Der Effekt $[\![k]\!]^\sharp$ einer Kante $k = (u, lab, v)$ hängt nur vom Label lab ab, d.h. $[\![k]\!]^\sharp = [\![lab]\!]^\sharp$

... die wir zur Ermittlung des Effekts eines Pfads $\pi = k_1 \ldots k_r$ zusammen setzen können:

$$[\![\pi]\!]^\sharp = [\![k_r]\!]^\sharp \circ \ldots \circ [\![k_1]\!]^\sharp$$

Der Effekt $[\![k]\!]^\sharp$ einer Kante $k = (u, lab, v)$ hängt nur vom Label *lab* ab, d.h. $[\![k]\!]^\sharp = [\![lab]\!]^\sharp$ wobei:

$$\begin{array}{lll} [\![;]\!]^\sharp\, A & = & A \\ [\![Pos(e)]\!]^\sharp\, A & = & [\![Neg(e)]\!]^\sharp\, A \qquad = \quad A \cup \{e\} \\ [\![R = e;]\!]^\sharp\, A & = & (A \cup \{e\}) \backslash Expr_R \qquad \text{wobei} \end{array}$$

$Expr_R$ alle Ausdrücke sind, die R enthalten

$$
\begin{aligned}
[\![R_1 = M[R_2];]\!]^\sharp\, A &= A \backslash Expr_{R_1} \\
[\![M[R_1] = R_2;]\!]^\sharp\, A &= A
\end{aligned}
$$

$$\begin{array}{lll} [\![R_1 = M[R_2];]\!]^\sharp\, A & = & A \backslash Expr_{R_1} \\ [\![M[R_1] = R_2;]\!]^\sharp\, A & = & A \end{array}$$

Damit können wir jeden Pfad untersuchen :-)

In einem Programm kann es mehrere Pfade geben :-(

Bei jeder Eingabe kann ein anderer gewählt werden :-((

$$
\begin{array}{lll}
[\![R_1 = M[R_2];]\!]^\sharp\, A & = & A \backslash Expr_{R_1} \\
[\![M[R_1] = R_2;]\!]^\sharp\, A & = & A
\end{array}
$$

Damit können wir jeden Pfad untersuchen :-)

In einem Programm kann es mehrere Pfade geben :-(

Bei jeder Eingabe kann ein anderer gewählt werden :-((

$\Longrightarrow$ Wir benötigen die Menge:

$$
\mathcal{A}[v] \quad = \quad \bigcap \{ [\![\pi]\!]^\sharp \emptyset \mid \pi : start \rightarrow^* v \}
$$

Im Klartext:

- $\longrightarrow$ Wir betrachten sämtliche Pfade, die v erreichen.
- $\longrightarrow$ Für jeden Pfad π bestimmen wir die Menge der entlang π verfügbaren Ausdrücke.
- $\longrightarrow$ Vor Programm-Ausführung ist nichts verfügbar :-)
- $\longrightarrow$ Wir bilden den Durchschnitt $\Longrightarrow$ sichere Information

Im Klartext:

$\longrightarrow$ Wir betrachten sämtliche Pfade, die v erreichen.

$\longrightarrow$ Für jeden Pfad π bestimmen wir die Menge der entlang π verfügbaren Ausdrücke.

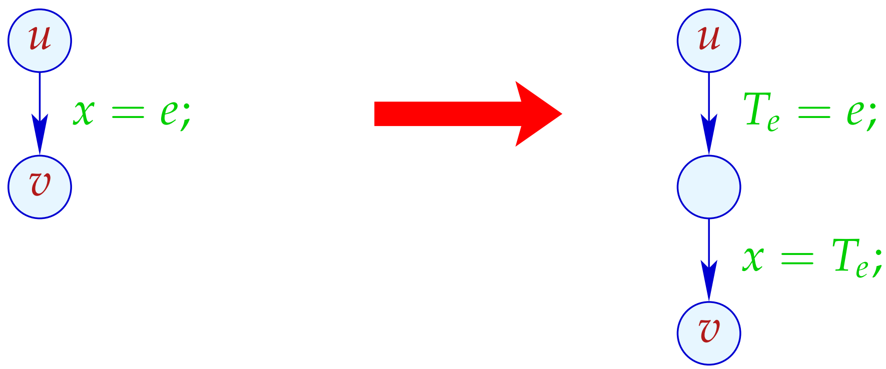
$\longrightarrow$ Vor Programm-Ausführung ist nichts verfügbar :-)

$\longrightarrow$ Wir bilden den Durchschnitt $\Longrightarrow$ sichere Information

Wie nutzen wir diese Information aus ???

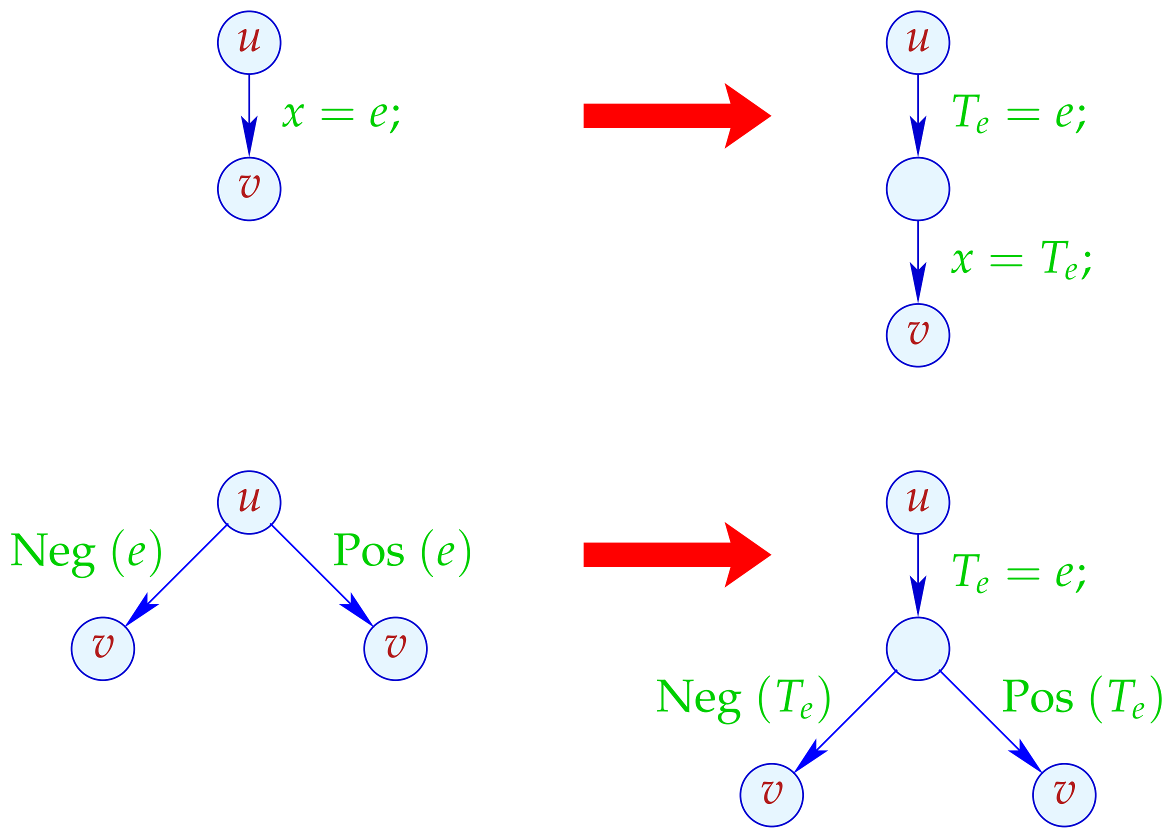
Transformation 1:

Wir stellen neue Register T_e als Speicherplatz für die e bereit:



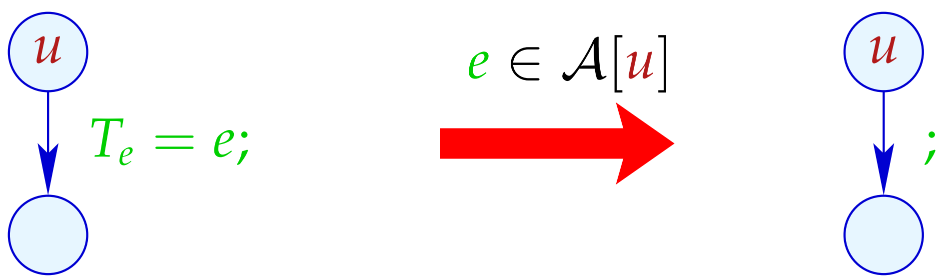
Transformation 1:

Wir stellen neue Register T_e als Speicherplatz für die e bereit:



Transformation 2:

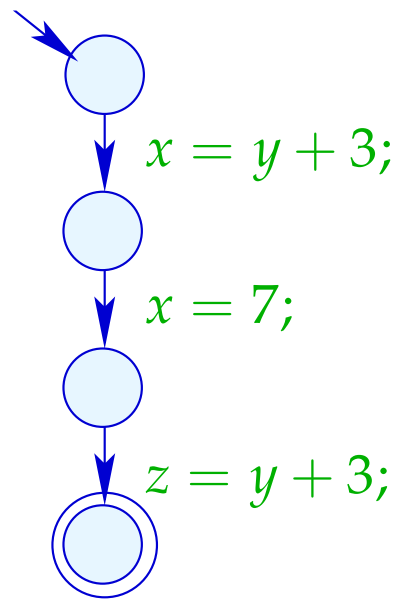
Falls e am Punkt u verfügbar ist, wird e nicht neu berechnet:



Wir ersetzen dann die Zuweisung durch *Nop* :-)

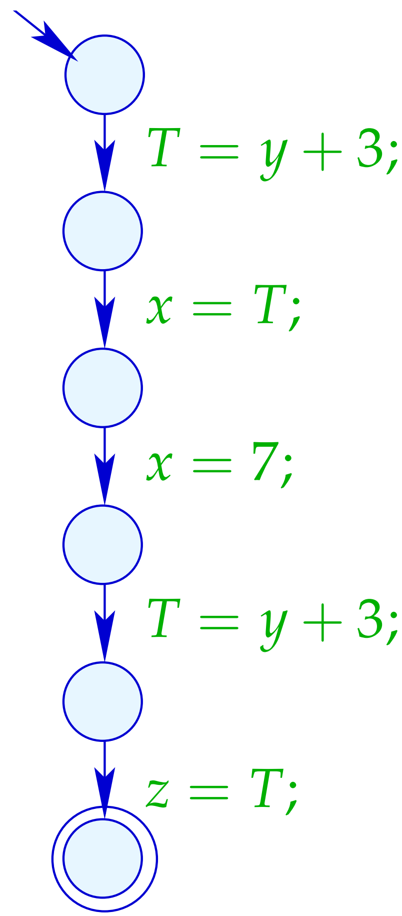
Beispiel:

$$
\begin{aligned}
x &= y + 3; \\
x &= 7; \\
z &= y + 3;
\end{aligned}
$$



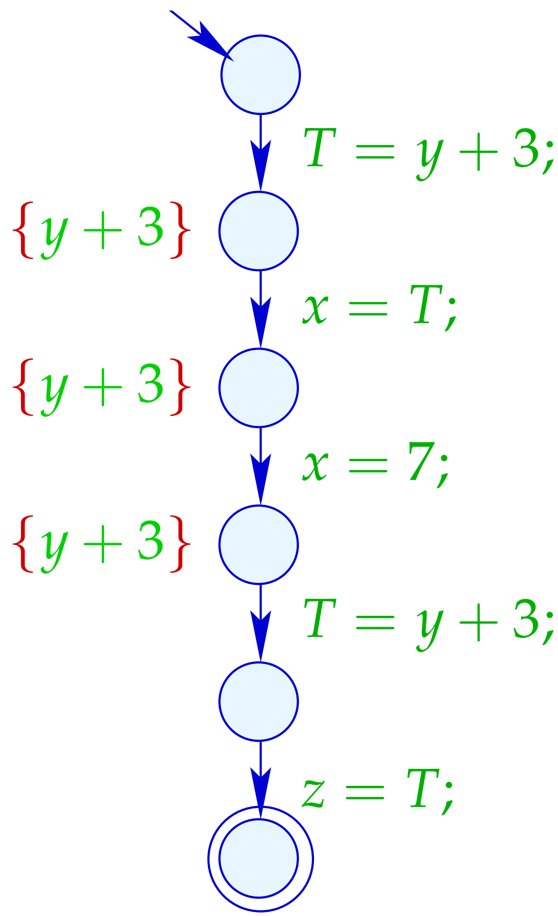
Beispiel:

$$\begin{aligned} x &= y + 3; \\ x &= 7; \\ z &= y + 3; \end{aligned}$$



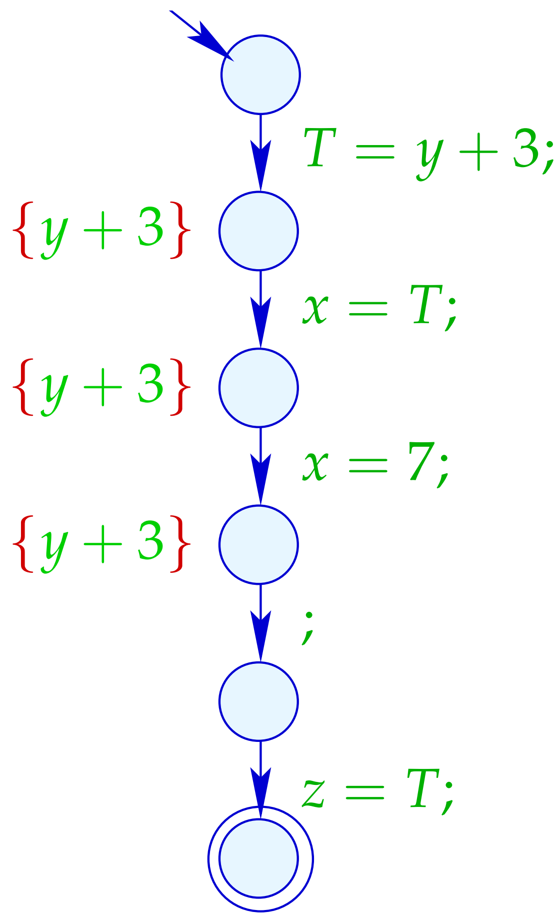
Beispiel:

$$
\begin{aligned}
x &= y + 3; \\
x &= 7; \\
z &= y + 3;
\end{aligned}
$$



Beispiel:

$$\begin{aligned} x &= y+3; \\ x &= 7; \\ z &= y+3; \end{aligned}$$



Korrektheit: (Idee)

Transformation 1 erhält offenbar die Bedeutung und $\mathcal{A}[u]$ für alle Knoten u :-)

Sei $\pi : start \rightarrow^* u$ der Pfad, den eine Berechnung nimmt.

Ist $e \in \mathcal{A}[u]$, dann auch $e \in [\![\pi]\!]^\sharp\, \emptyset$.

Dann muss es eine Zerlegung von π geben:

$$\text{start} \overset{\pi_1}{\rightsquigarrow} u_1 \xrightarrow{k} u_2 \overset{\pi_2}{\rightsquigarrow} u$$

mit den folgenden Eigenschaften:

- Der Ausdruck e wird an der Kante k berechnet;
- Der Ausdruck e wird an keiner Kante in π_2 aus der Menge der verfügbaren Ausdrücke entfernt, d.h. keine Variable von e erhält einen neuen Wert :-)

- Der Ausdruck e wird an der Kante k berechnet;
- Der Ausdruck e wird an keiner Kante in π_2 aus der Menge der verfügbaren Ausdrücke entfernt, d.h. keine Variable von e erhält einen neuen Wert :-)

$\Longrightarrow$

Wird u erreicht, enthält das Register T_e den Wert von e :-))

Achtung:

Die Transformation 1 ist nur sinnvoll an Zuweisungen $x = e;$, wobei:

$\longrightarrow$ $x \notin Vars(e)$;

$\longrightarrow$ $e \notin Vars$;

$\longrightarrow$ sich die Berechnung von e lohnt :-}

Achtung:

Die Transformation 1 ist nur sinnvoll an Zuweisungen $x = e;$, wobei:

$\rightarrow$ $x \notin Vars(e)$;

$\rightarrow$ $e \notin Vars$;

$\rightarrow$ sich die Berechnung von e lohnt :-}

Bleibt die Preisfrage ...

Preisfrage:

Wie berechnen wir $\mathcal{A}[u]$ für jeden Programmpunkt u ??

Preisfrage:

Wie berechnen wir $\mathcal{A}[u]$ für jeden Programmpunkt u ??

Idee:

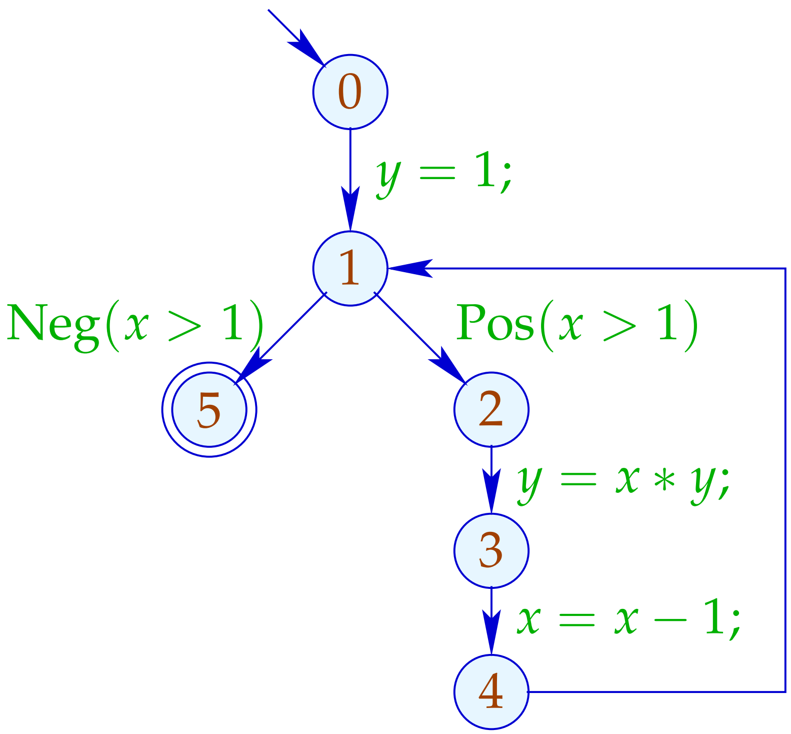
Wir stellen ein Constraint-System auf, das alle Bedingungen an die Werte $\mathcal{A}[u]$ sammelt:

$$\begin{array}{lll} \mathcal{A}[\textit{start}] & \subseteq & \emptyset \\ \mathcal{A}[v] & \subseteq & [\![k]\!]^\sharp\,(\mathcal{A}[u]) \qquad\qquad k = (u, \_, v) \quad \text{Kante} \end{array}$$

Gesucht:

- möglichst große Lösung (??)
- Algorithmus, der diese berechnet :-)

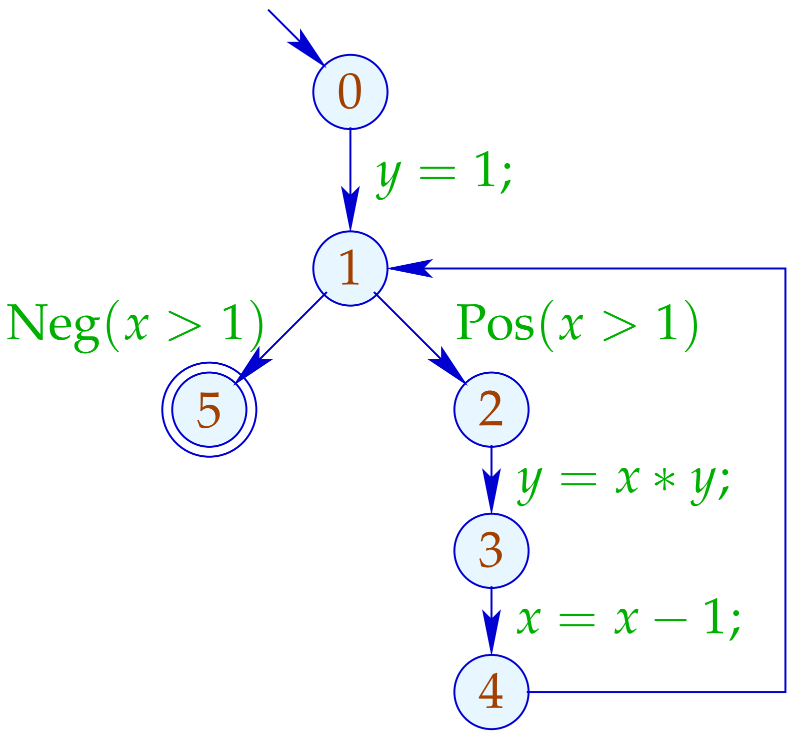
Beispiel:



Gesucht:

- möglichst große Lösung (??)
- Algorithmus, der diese berechnet :-)

Beispiel:

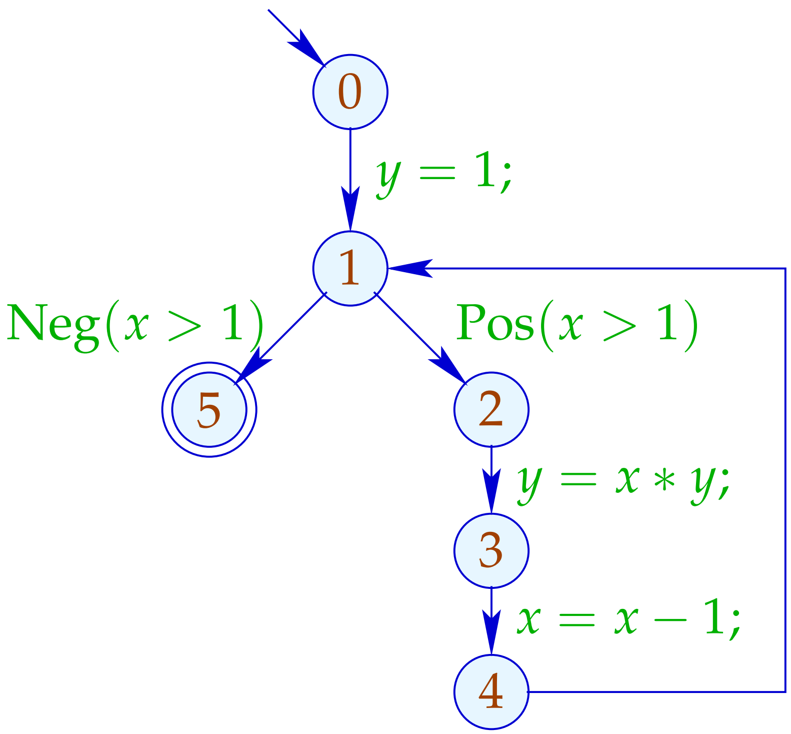


$$\mathcal{A}[0] \quad \subseteq \quad \emptyset$$

Gesucht:

- möglichst große Lösung (??)
- Algorithmus, der diese berechnet :-)

Beispiel:

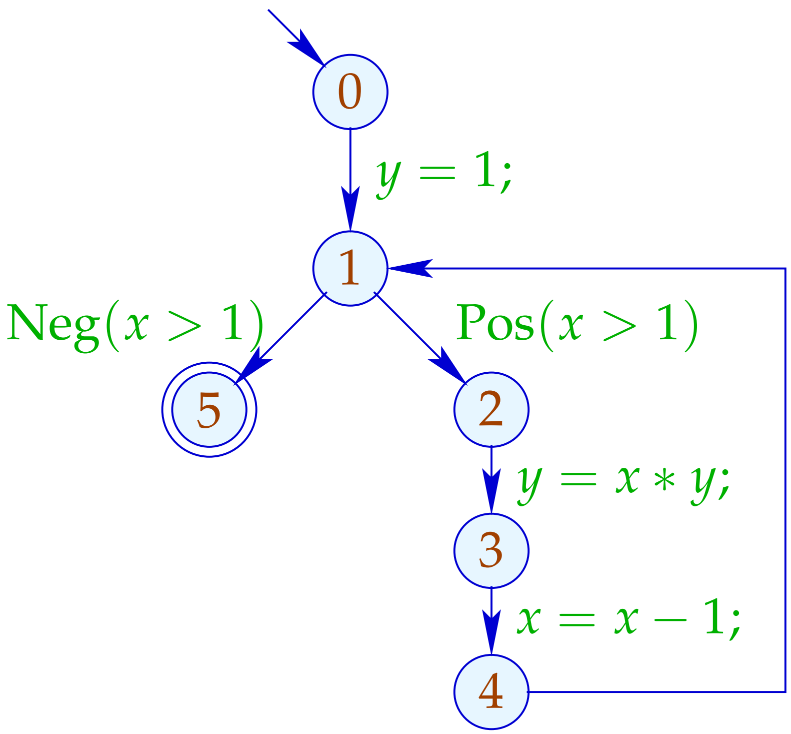


$$
\begin{array}{lll}
\mathcal{A}[0] & \subseteq & \emptyset \\
\mathcal{A}[1] & \subseteq & (\mathcal{A}[0] \cup \{1\}) \backslash \mathit{Expr}_y \\
\mathcal{A}[1] & \subseteq & \mathcal{A}[4]
\end{array}
$$

Gesucht:

- möglichst große Lösung (??)
- Algorithmus, der diese berechnet :-)

Beispiel:

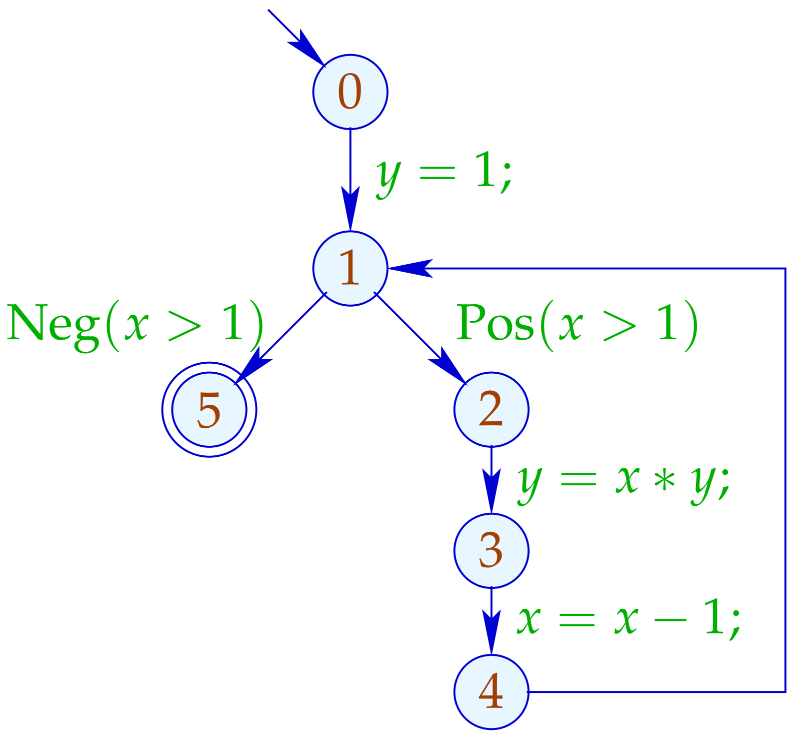


$$\begin{array}{lcl}
\mathcal{A}[0] & \subseteq & \emptyset \\
\mathcal{A}[1] & \subseteq & (\mathcal{A}[0] \cup \{1\}) \backslash Expr_y \\
\mathcal{A}[1] & \subseteq & \mathcal{A}[4] \\
\mathcal{A}[2] & \subseteq & \mathcal{A}[1] \cup \{x > 1\}
\end{array}$$

Gesucht:

- möglichst **große** Lösung (??)
- Algorithmus, der diese berechnet :-)

Beispiel:

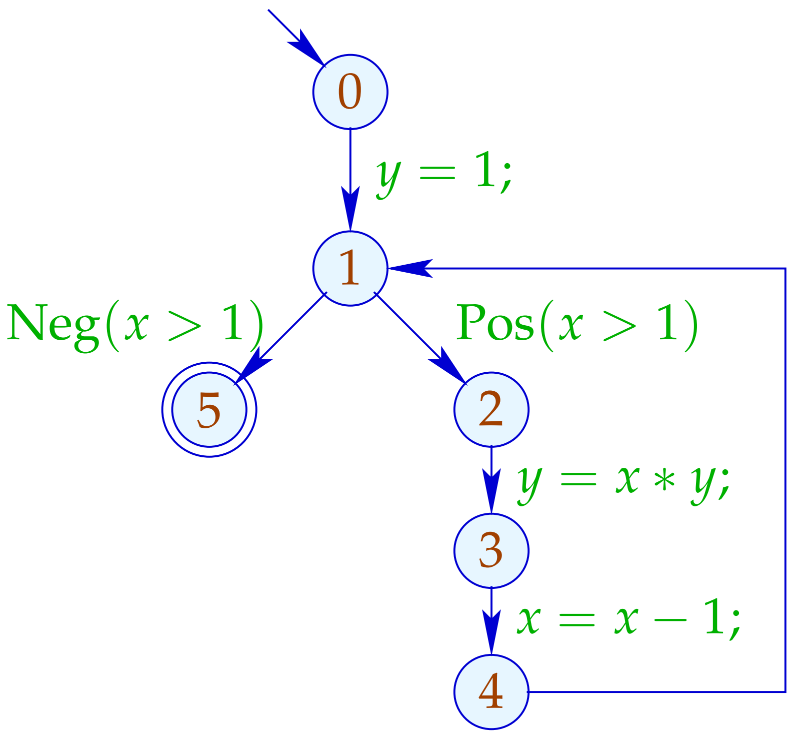


$$
\begin{array}{lll}
\mathcal{A}[0] & \subseteq & \emptyset \\
\mathcal{A}[1] & \subseteq & (\mathcal{A}[0] \cup \{1\}) \backslash Expr_y \\
\mathcal{A}[1] & \subseteq & \mathcal{A}[4] \\
\mathcal{A}[2] & \subseteq & \mathcal{A}[1] \cup \{x > 1\} \\
\mathcal{A}[3] & \subseteq & (\mathcal{A}[2] \cup \{x * y\}) \backslash Expr_y
\end{array}
$$

Gesucht:

- möglichst große Lösung (??)
- Algorithmus, der diese berechnet :-)

Beispiel:

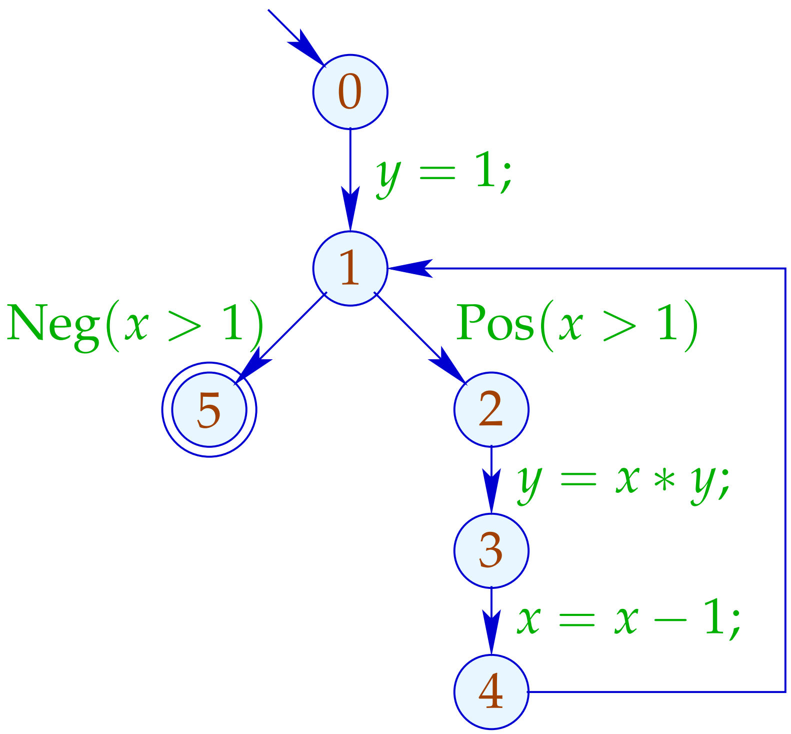


$$
\begin{array}{lll}
\mathcal{A}[0] & \subseteq & \emptyset \\
\mathcal{A}[1] & \subseteq & (\mathcal{A}[0] \cup \{1\}) \backslash \mathit{Expr}_y \\
\mathcal{A}[1] & \subseteq & \mathcal{A}[4] \\
\mathcal{A}[2] & \subseteq & \mathcal{A}[1] \cup \{x > 1\} \\
\mathcal{A}[3] & \subseteq & (\mathcal{A}[2] \cup \{x * y\}) \backslash \mathit{Expr}_y \\
\mathcal{A}[4] & \subseteq & (\mathcal{A}[3] \cup \{x - 1\}) \backslash \mathit{Expr}_x
\end{array}
$$

Gesucht:

- möglichst große Lösung (??)
- Algorithmus, der diese berechnet :-)

Beispiel:

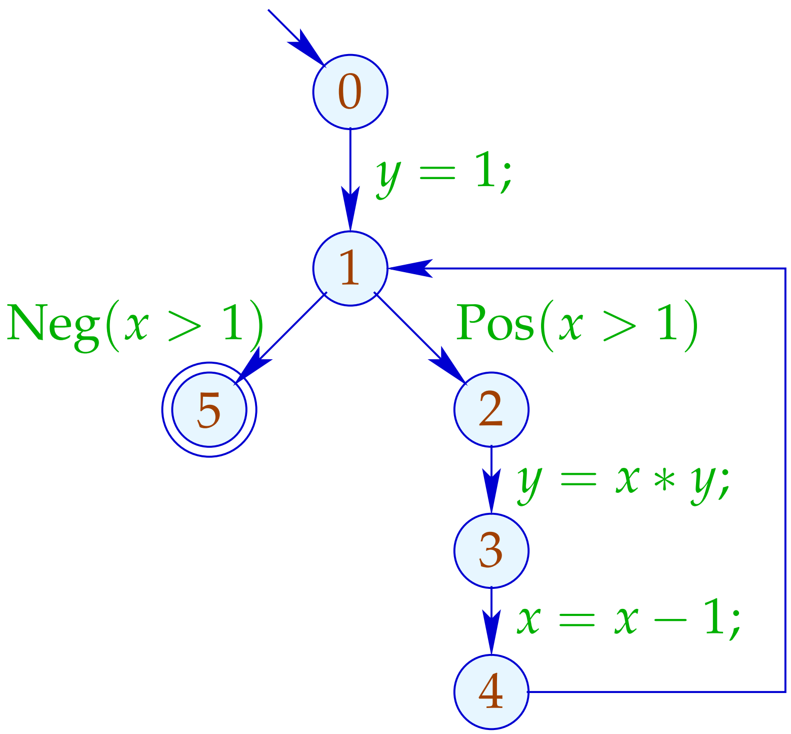


$$
\begin{array}{lll}
\mathcal{A}[0] & \subseteq & \emptyset \\
\mathcal{A}[1] & \subseteq & (\mathcal{A}[0] \cup \{1\}) \backslash Expr_y \\
\mathcal{A}[1] & \subseteq & \mathcal{A}[4] \\
\mathcal{A}[2] & \subseteq & \mathcal{A}[1] \cup \{x > 1\} \\
\mathcal{A}[3] & \subseteq & (\mathcal{A}[2] \cup \{x * y\}) \backslash Expr_y \\
\mathcal{A}[4] & \subseteq & (\mathcal{A}[3] \cup \{x - 1\}) \backslash Expr_x \\
\mathcal{A}[5] & \subseteq & \mathcal{A}[1] \cup \{x > 1\}
\end{array}
$$

Gesucht:

- möglichst große Lösung (??)
- Algorithmus, der diese berechnet :-)

Beispiel:



Lösung:

$$
\begin{aligned}
\mathcal{A}[0] &= \emptyset \\
\mathcal{A}[1] &= \{1\} \\
\mathcal{A}[2] &= \{1, x > 1\} \\
\mathcal{A}[3] &= \{1, x > 1\} \\
\mathcal{A}[4] &= \{1\} \\
\mathcal{A}[5] &= \{1, x > 1\}
\end{aligned}
$$

Beobachtung:

- Die möglichen Werte für $\mathcal{A}[u]$ bilden einen vollständigen Verband:

$$\mathbb{D} = 2^{Expr} \quad \text{mit} \quad B_1 \sqsubseteq B_2 \quad \text{gdw.} \quad B_1 \supseteq B_2$$

Beobachtung:

- Die möglichen Werte für $\mathcal{A}[u]$ bilden einen vollständigen Verband:

$$\mathbb{D} = 2^{Expr} \quad \text{mit} \quad B_1 \sqsubseteq B_2 \quad \text{gdw.} \quad B_1 \supseteq B_2$$

- Die Funktionen $[\![k]\!]^\sharp : \mathbb{D} \to \mathbb{D}$ sind monoton, d.h.

$$[\![k]\!]^\sharp(B_1) \sqsubseteq [\![k]\!]^\sharp(B_2) \quad \text{gdw.} \quad B_1 \sqsubseteq B_2$$

Exkurs 2: Vollständige Verbände

Eine Menge $\mathbb{D}$ mit einer Relation $\sqsubseteq \subseteq \mathbb{D} \times \mathbb{D}$ ist eine partielle Ordnung falls für alle $a, b, c \in \mathbb{D}$ gilt:

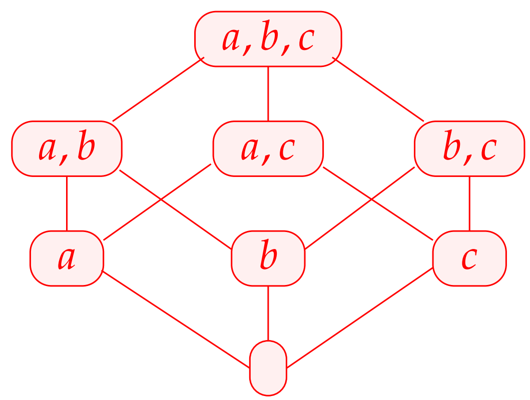
$$a \sqsubseteq a \qquad \textit{Reflexivität}$$

$$a \sqsubseteq b \wedge b \sqsubseteq a \implies a = b \qquad \textit{Anti-Symmetrie}$$

$$a \sqsubseteq b \wedge b \sqsubseteq c \implies a \sqsubseteq c \qquad \textit{Transitivität}$$

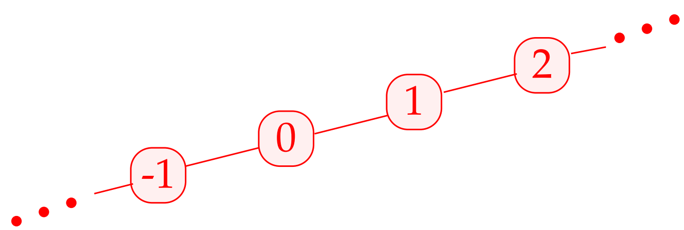
Beispiele:

1. $\mathbb{D} = 2^{\{a,b,c\}}$ mit der Relation "$\subseteq$" :

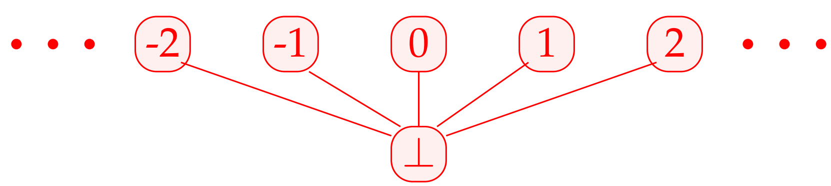


3. $\mathbb{Z}$ mit der Relation "$=$" :

3. $\mathbb{Z}$ mit der Relation "$\leq$" :



4. $\mathbb{Z}_\perp = \mathbb{Z} \cup \{\perp\}$ mit der Ordnung:



$d \in \mathbb{D}$ heißt obere Schranke für $X \subseteq \mathbb{D}$ falls

$$x \sqsubseteq d \qquad \text{für alle } x \in X$$

$d \in \mathbb{D}$ heißt obere Schranke für $X \subseteq \mathbb{D}$ falls

$$x \sqsubseteq d \qquad \text{für alle } x \in X$$

d heißt kleinste obere Schranke (lub) falls

1. d eine obere Schranke ist und
2. $d \sqsubseteq y$ für jede obere Schranke y für X.

$d \in \mathbb{D}$ heißt obere Schranke für $X \subseteq \mathbb{D}$ falls

$$x \sqsubseteq d \qquad \text{für alle } x \in X$$

d heißt kleinste obere Schranke (lub) falls

1. d eine obere Schranke ist und
2. $d \sqsubseteq y$ für jede obere Schranke y für X.

Achtung:

- $\{0, 2, 4, \ldots\} \subseteq \mathbb{Z}$ besitzt keine obere Schranke!
- $\{0, 2, 4\} \subseteq \mathbb{Z}$ besitzt die oberen Schranken $4, 5, 6, \ldots$

Ein vollständiger Verband (cl) $\mathbb{D}$ ist eine partielle Ordnung, in der jede Teilmenge $X \subseteq \mathbb{D}$ eine kleinste obere Schranke $\bigsqcup X \in \mathbb{D}$ besitzt.

Beachte:

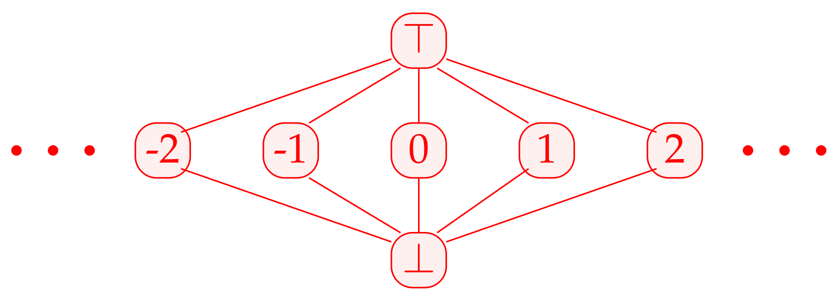
Jeder vollständige Verband besitzt

→ ein kleinstes Element $\bot = \bigsqcup \emptyset \quad \in \mathbb{D}$;

→ ein größtes Element $\top = \bigsqcup \mathbb{D} \quad \in \mathbb{D}$.

Beispiele:

1. $\mathbb{D} = 2^{\{a,b,c\}}$ ist ein cl   :-)
2. $\mathbb{D} = \mathbb{Z}$ mit "$=$" ist keiner.
3. $\mathbb{D} = \mathbb{Z}$ mit "$\leq$" ebenfalls nicht.
4. $\mathbb{D} = \mathbb{Z}_\perp$ auch nicht   :-(
5. Mit einem zusätzlichen Symbol $\top$ erhalten wir den flachen Verband $\mathbb{Z}_\perp^\top = \mathbb{Z} \cup \{\perp, \top\}$:



Es gilt:

Satz:

In jedem vollständigen Verband $\mathbb{D}$ besitzt jede Teilmenge $X \subseteq \mathbb{D}$ eine größte untere Schranke $\bigsqcap X$.

Es gilt:

Satz:

In jedem vollständigen Verband $\mathbb{D}$ besitzt jede Teilmenge $X \subseteq \mathbb{D}$ eine größte untere Schranke $\bigsqcap X$.

Beweis:

Konstruiere $U = \{u \in \mathbb{D} \mid \forall\, x \in X : \; u \sqsubseteq x\}$.

// die Menge der unteren Schranken von X :-)

Es gilt:

Satz:

In jedem vollständigen Verband $\mathbb{D}$ besitzt jede Teilmenge $X \subseteq \mathbb{D}$ eine größte untere Schranke $\bigsqcap X$.

Beweis:

Konstruiere $U = \{u \in \mathbb{D} \mid \forall\, x \in X : \; u \sqsubseteq x\}$.

// die Menge der unteren Schranken von X :-)

Setze: $g := \bigsqcup U$

Behauptung: $g = \bigsqcap X$

(1) g ist eine untere Schranke von X :

Für $\quad x \in X \quad$ gilt:

$$u \sqsubseteq x \text{ für alle } u \in U$$

$$\Longrightarrow \quad x \text{ ist obere Schranke von } U$$

$$\Longrightarrow \quad g \sqsubseteq x \qquad \text{:-)}$$

(1) g ist eine untere Schranke von X :

Für $x \in X$ gilt:

$$u \sqsubseteq x \text{ für alle } u \in U$$

$$\Longrightarrow \quad x \text{ ist obere Schranke von } U$$

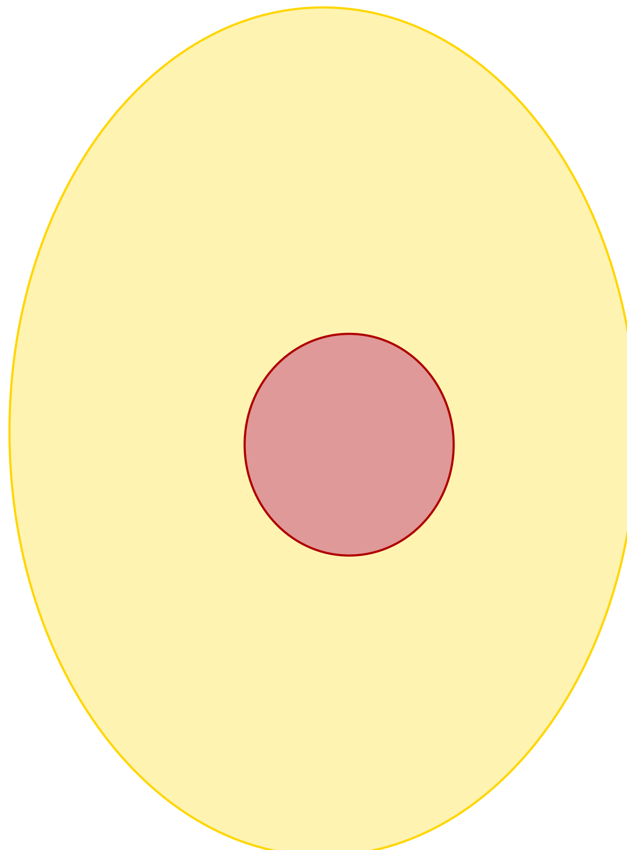
$$\Longrightarrow \quad g \sqsubseteq x \qquad \text{:-)}$$

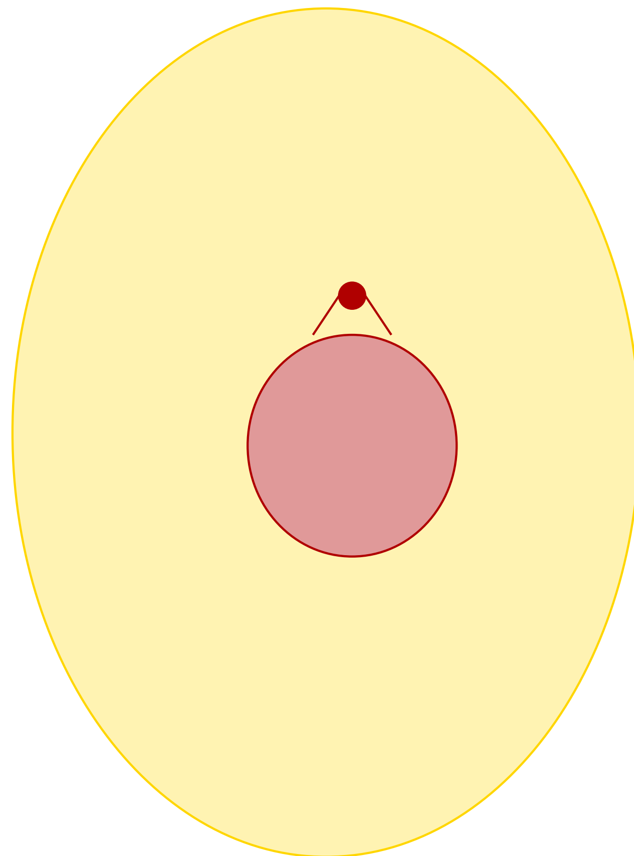
(2) g ist größte untere Schranke von X :

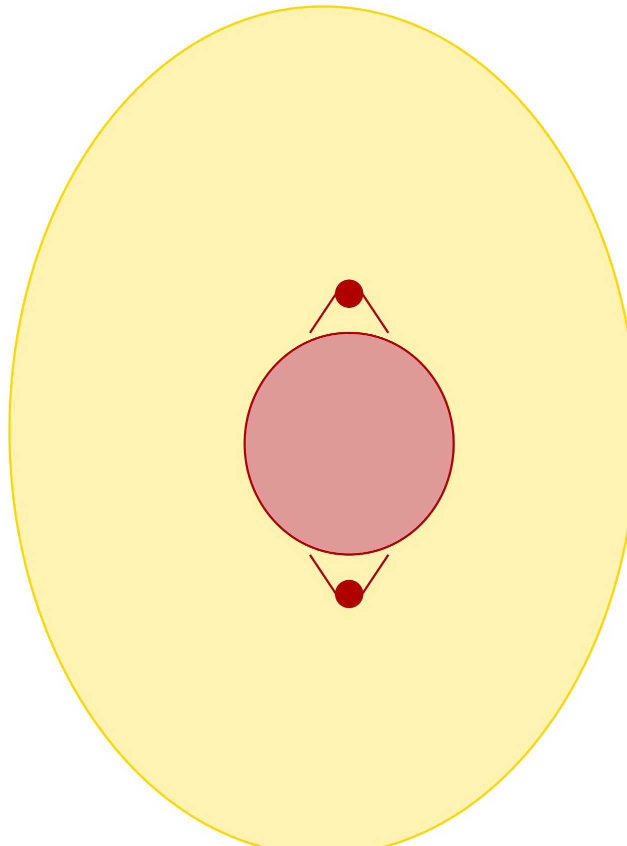
Für jede untere Schranke u von X gilt:

$$u \in U$$

$$\Longrightarrow \quad u \sqsubseteq g \qquad \text{:-))}$$







Wir suchen Lösungen für Constraint-Systeme der Form:

$$x_i \quad \sqsupseteq \quad f_i(x_1, \ldots, x_n) \qquad\qquad (*)$$

Wir suchen Lösungen für Constraint-Systeme der Form:

$$x_i \quad \sqsupseteq \quad f_i(x_1, \ldots, x_n) \qquad\qquad (*)$$

wobei:

x_i	Unbekannte	hier:	$\mathcal{A}[u]$
$\mathbb{D}$	Werte	hier:	2^{Expr}
$\sqsubseteq \quad \subseteq \quad \mathbb{D} \times \mathbb{D}$	Ordnungsrelation	hier:	$\supseteq$
$f_i\colon \mathbb{D}^n \longrightarrow \mathbb{D}$	Bedingung	hier:	...

Wir suchen Lösungen für Constraint-Systeme der Form:

$$x_i \quad \sqsupseteq \quad f_i(x_1, \ldots, x_n) \qquad\qquad (*)$$

wobei:

x_i	Unbekannte	hier:	$\mathcal{A}[u]$
$\mathbb{D}$	Werte	hier:	2^{Expr}
$\sqsubseteq \quad \subseteq \quad \mathbb{D} \times \mathbb{D}$	Ordnungsrelation	hier:	$\supseteq$
$f_i : \mathbb{D}^n \to \mathbb{D}$	Bedingung	hier:	...

Constraint für $\mathcal{A}[v]$:

$$\mathcal{A}[v] \quad \subseteq \quad \bigcap \{ [\![k]\!]^\sharp (\mathcal{A}[u]) \mid k = (u, \_, v) \ \text{Kante} \}$$

Wir suchen Lösungen für Constraint-Systeme der Form:

$$x_i \quad \sqsupseteq \quad f_i(x_1, \ldots, x_n) \qquad\qquad (*)$$

wobei:

x_i	Unbekannte	hier:	$\mathcal{A}[u]$
$\mathbb{D}$	Werte	hier:	2^{Expr}
$\sqsubseteq \quad \subseteq \quad \mathbb{D} \times \mathbb{D}$	Ordnungsrelation	hier:	$\supseteq$
$f_i \colon \mathbb{D}^n \to \mathbb{D}$	Bedingung	hier:	...

Constraint für $\mathcal{A}[v]$:

$$\mathcal{A}[v] \quad \subseteq \quad \bigcap\{[\![k]\!]^\sharp(\mathcal{A}[u]) \mid k = (u, \_, v) \text{ Kante}\}$$

Denn:

$$x \sqsupseteq d_1 \wedge \ldots \wedge x \sqsupseteq d_k \quad \text{gdw.} \quad x \sqsupseteq \bigsqcup\{d_1, \ldots, d_k\} \qquad \text{:-)}$$

Eine Abbildung $f : \mathbb{D}_1 \to \mathbb{D}_2$ heißt monoton, falls
$f(a) \sqsubseteq f(b)$ für alle $a \sqsubseteq b$.

Eine Abbildung $f : \mathbb{D}_1 \to \mathbb{D}_2$ heißt monoton, falls
$f(a) \sqsubseteq f(b)$ für alle $a \sqsubseteq b$.

Beispiele:

(1) $\mathbb{D}_1 = \mathbb{D}_2 = 2^U$ für eine Menge U und $f\, x = (x \cap a) \cup b$.
Offensichtlich ist jedes solche f monoton :-)

Eine Abbildung $f : \mathbb{D}_1 \to \mathbb{D}_2$ heißt monoton, falls $f(a) \sqsubseteq f(b)$ für alle $a \sqsubseteq b$.

Beispiele:

(1) $\mathbb{D}_1 = \mathbb{D}_2 = 2^U$ für eine Menge U und $f\,x = (x \cap a) \cup b$.
Offensichtlich ist jedes solche f monoton :-)

(2) $\mathbb{D}_1 = \mathbb{D}_2 = \mathbb{Z}$ (mit der Ordnung "$\leq$"). Dann gilt:

- $\mathsf{inc}\,x = x + 1$ ist monoton.
- $\mathsf{dec}\,x = x - 1$ ist monoton.

Eine Abbildung $f : \mathbb{D}_1 \to \mathbb{D}_2$ heißt monoton, falls
$f(a) \sqsubseteq f(b)$ für alle $a \sqsubseteq b$.

Beispiele:

(1) $\mathbb{D}_1 = \mathbb{D}_2 = 2^U$ für eine Menge U und $f\,x = (x \cap a) \cup b$.
Offensichtlich ist jedes solche f monoton :-)

(2) $\mathbb{D}_1 = \mathbb{D}_2 = \mathbb{Z}$ (mit der Ordnung "$\leq$"). Dann gilt:

- $\mathsf{inc}\,x = x + 1$ ist monoton.
- $\mathsf{dec}\,x = x - 1$ ist monoton.
- $\mathsf{inv}\,x = -x$ ist nicht monoton :-)

Satz:

Sind $f_1 : \mathbb{D}_1 \to \mathbb{D}_2$ und $f_2 : \mathbb{D}_2 \to \mathbb{D}_3$ monoton, dann ist auch $f_2 \circ f_1 : \mathbb{D}_1 \to \mathbb{D}_3$ monoton :-)

Satz:

Sind $f_1 : \mathbb{D}_1 \to \mathbb{D}_2$ und $f_2 : \mathbb{D}_2 \to \mathbb{D}_3$ monoton, dann ist auch $f_2 \circ f_1 : \mathbb{D}_1 \to \mathbb{D}_3$ monoton :-)

Satz:

Ist $\mathbb{D}_2$ ein vollständiger Verband, dann bildet auch die Menge $[\mathbb{D}_1 \to \mathbb{D}_2]$ der monotonen Funktionen $f : \mathbb{D}_1 \to \mathbb{D}_2$ einen vollständigen Verband, wobei

$$f \sqsubseteq g \quad \text{gdw.} \quad f\,x \sqsubseteq g\,x \quad \text{für alle } x \in \mathbb{D}_1$$

Satz:

Sind $f_1 : \mathbb{D}_1 \to \mathbb{D}_2$ und $f_2 : \mathbb{D}_2 \to \mathbb{D}_3$ monoton, dann ist auch $f_2 \circ f_1 : \mathbb{D}_1 \to \mathbb{D}_3$ monoton :-)

Satz:

Ist $\mathbb{D}_2$ ein vollständiger Verband, dann bildet auch die Menge $[\mathbb{D}_1 \to \mathbb{D}_2]$ der monotonen Funktionen $f : \mathbb{D}_1 \to \mathbb{D}_2$ einen vollständigen Verband, wobei

$$f \sqsubseteq g \quad \text{gdw.} \quad f\,x \sqsubseteq g\,x \quad \text{für alle } x \in \mathbb{D}_1$$

Insbesondere ist für $F \subseteq [\mathbb{D}_1 \to \mathbb{D}_2]$,

$$\bigsqcup F = f \quad \text{mit} \quad f\,x = \bigsqcup \{g\,x \mid g \in F\}$$

Für Funktionen $f_i\, x = a_i \cap x \cup b_i$ können wir die Operationen "$\circ$", "$\sqcup$" und "$\sqcap$" explizit angeben:

$$
\begin{array}{lll}
(f_2 \circ f_1)\, x & = & \boxed{a_1 \cap a_2} \cap x \cup \boxed{a_2 \cap b_1 \cup b_2} \\
(f_1 \sqcup f_2)\, x & = & \boxed{(a_1 \cup a_2)} \cap x \cup \boxed{b_1 \cup b_2} \\
(f_1 \sqcap f_2)\, x & = & \boxed{(a_1 \cup b_1) \cap (a_2 \cup b_2)} \cap x \cup \boxed{b_1 \cap b_2}
\end{array}
$$

Gesucht: möglichst kleine Lösung für:

$$x_i \sqsupseteq f_i(x_1, \ldots, x_n), \quad i = 1, \ldots, n \qquad (*)$$

wobei alle $f_i : \mathbb{D}^n \to \mathbb{D}$ monoton sind.

Gesucht: möglichst kleine Lösung für:

$$x_i \sqsupseteq f_i(x_1, \ldots, x_n), \quad i = 1, \ldots, n \qquad (*)$$

wobei alle $f_i : \mathbb{D}^n \to \mathbb{D}$ monoton sind.

Idee:

- Betrachte $F : \mathbb{D}^n \to \mathbb{D}^n$ mit

  $F(x_1, \ldots, x_n) = (y_1, \ldots, y_n)$ wobei $y_i = f_i(x_1, \ldots, x_n)$.

Gesucht: möglichst kleine Lösung für:

$$x_i \sqsupseteq f_i(x_1, \ldots, x_n), \quad i = 1, \ldots, n \qquad (*)$$

wobei alle $f_i : \mathbb{D}^n \to \mathbb{D}$ monoton sind.

Idee:

- Betrachte $F : \mathbb{D}^n \to \mathbb{D}^n$ mit

$$F(x_1, \ldots, x_n) = (y_1, \ldots, y_n) \quad \text{wobei} \quad y_i = f_i(x_1, \ldots, x_n).$$

- Sind alle f_i monoton, dann auch F :-)

Gesucht: möglichst kleine Lösung für:

$$x_i \sqsupseteq f_i(x_1, \ldots, x_n), \quad i = 1, \ldots, n \qquad (*)$$

wobei alle $f_i : \mathbb{D}^n \to \mathbb{D}$ monoton sind.

Idee:

- Betrachte $F : \mathbb{D}^n \to \mathbb{D}^n$ mit

$$F(x_1, \ldots, x_n) = (y_1, \ldots, y_n) \quad \text{wobei} \quad y_i = f_i(x_1, \ldots, x_n).$$

- Sind alle f_i monoton, dann auch F :-)
- Wir approximieren sukzessive eine Lösung. Wir konstruieren:

$$\underline{\perp}, \quad F\,\underline{\perp}, \quad F^2\,\underline{\perp}, \quad F^3\,\underline{\perp}, \quad \ldots$$

Hoffnung: Wir erreichen irgendwann eine Lösung ... ???

Beispiel: $\mathbb{D} = 2^{\{a,b,c\}}, \quad \sqsubseteq \; = \; \subseteq$

$$\begin{array}{lll} x_1 & \supseteq & \{a\} \cup x_3 \\ x_2 & \supseteq & x_3 \cap \{a, b\} \\ x_3 & \supseteq & x_1 \cup \{c\} \end{array}$$

Beispiel: $\mathbb{D} = 2^{\{a,b,c\}}, \quad \sqsubseteq \, = \, \subseteq$

$$\begin{array}{lll} x_1 & \supseteq & \{a\} \cup x_3 \\ x_2 & \supseteq & x_3 \cap \{a, b\} \\ x_3 & \supseteq & x_1 \cup \{c\} \end{array}$$

Die Iteration:

	0	1	2	3	4
x_1	$\emptyset$				
x_2	$\emptyset$				
x_3	$\emptyset$				

Beispiel: $\mathbb{D} = 2^{\{a,b,c\}}, \quad \sqsubseteq = \subseteq$

$$\begin{array}{lll} x_1 & \supseteq & \{a\} \cup x_3 \\ x_2 & \supseteq & x_3 \cap \{a, b\} \\ x_3 & \supseteq & x_1 \cup \{c\} \end{array}$$

Die Iteration:

	0	1	2	3	4
x_1	$\emptyset$	$\{a\}$			
x_2	$\emptyset$	$\emptyset$			
x_3	$\emptyset$	$\{c\}$			

Beispiel: $\mathbb{D} = 2^{\{a,b,c\}}, \quad \sqsubseteq\ =\ \subseteq$

$$\begin{array}{lll} x_1 & \supseteq & \{a\} \cup x_3 \\ x_2 & \supseteq & x_3 \cap \{a, b\} \\ x_3 & \supseteq & x_1 \cup \{c\} \end{array}$$

Die Iteration:

	0	1	2	3	4
x_1	$\emptyset$	$\{a\}$	$\{a, c\}$		
x_2	$\emptyset$	$\emptyset$	$\emptyset$		
x_3	$\emptyset$	$\{c\}$	$\{a, c\}$		

Beispiel: $\mathbb{D} = 2^{\{a,b,c\}}, \quad \sqsubseteq \; = \; \subseteq$

$$\begin{array}{lll} x_1 & \supseteq & \{a\} \cup x_3 \\ x_2 & \supseteq & x_3 \cap \{a,b\} \\ x_3 & \supseteq & x_1 \cup \{c\} \end{array}$$

Die Iteration:

	0	1	2	3	4
x_1	$\emptyset$	$\{a\}$	$\{a,c\}$	$\{a,c\}$	
x_2	$\emptyset$	$\emptyset$	$\emptyset$	$\{a\}$	
x_3	$\emptyset$	$\{c\}$	$\{a,c\}$	$\{a,c\}$	

Beispiel: $\mathbb{D} = 2^{\{a,b,c\}}, \quad \sqsubseteq \; = \; \subseteq$

$$\begin{array}{lll} x_1 & \supseteq & \{a\} \cup x_3 \\ x_2 & \supseteq & x_3 \cap \{a, b\} \\ x_3 & \supseteq & x_1 \cup \{c\} \end{array}$$

Die Iteration:

	0	1	2	3	4
x_1	$\emptyset$	$\{a\}$	$\{a, c\}$	$\{a, c\}$	dito
x_2	$\emptyset$	$\emptyset$	$\emptyset$	$\{a\}$	
x_3	$\emptyset$	$\{c\}$	$\{a, c\}$	$\{a, c\}$	

Offenbar gilt:

- Gilt $F^k \underline{\perp} = F^{k+1} \underline{\perp}$, ist eine Lösung gefunden :-)
- $\underline{\perp}, F\,\underline{\perp}, F^2\,\underline{\perp}, \ldots$ bilden eine aufsteigende Kette :

$$\underline{\perp} \quad \sqsubseteq \quad F\,\underline{\perp} \quad \sqsubseteq \quad F^2\,\underline{\perp} \quad \sqsubseteq \quad \ldots$$

- Sind alle aufsteigenden Ketten endlich, gibt es k immer.

Offenbar gilt:

- Gilt $F^k \underline{\perp} = F^{k+1} \underline{\perp}$, ist eine Lösung gefunden :-)
- $\underline{\perp}, F\, \underline{\perp}, F^2\, \underline{\perp}, \ldots$ bilden eine aufsteigende Kette :

$$\underline{\perp} \quad \sqsubseteq \quad F\, \underline{\perp} \quad \sqsubseteq \quad F^2\, \underline{\perp} \quad \sqsubseteq \quad \ldots$$

- Sind alle aufsteigenden Ketten endlich, gibt es k immer.

Die zweite Aussage folgt mit vollständiger Induktion:

Offenbar gilt:

- Gilt $F^k \underline{\perp} = F^{k+1} \underline{\perp}$, ist eine Lösung gefunden :-)
- $\underline{\perp}, F\,\underline{\perp}, F^2\,\underline{\perp}, \ldots$ bilden eine aufsteigende Kette :

$$\underline{\perp} \quad \sqsubseteq \quad F\,\underline{\perp} \quad \sqsubseteq \quad F^2\,\underline{\perp} \quad \sqsubseteq \quad \ldots$$

- Sind alle aufsteigenden Ketten endlich, gibt es k immer.

Die zweite Aussage folgt mit vollständiger Induktion:

Anfang: $F^0\,\underline{\perp} = \underline{\perp} \sqsubseteq F^1\,\underline{\perp}$:-)

Offenbar gilt:

- Gilt $F^k \underline{\perp} = F^{k+1} \underline{\perp}$, ist eine Lösung gefunden :-)
- $\underline{\perp}, F \underline{\perp}, F^2 \underline{\perp}, \ldots$ bilden eine aufsteigende Kette :

$$\underline{\perp} \quad \sqsubseteq \quad F \underline{\perp} \quad \sqsubseteq \quad F^2 \underline{\perp} \quad \sqsubseteq \quad \ldots$$

- Sind alle aufsteigenden Ketten endlich, gibt es k immer.

Die zweite Aussage folgt mit vollständiger Induktion:

Anfang: $F^0 \underline{\perp} = \underline{\perp} \sqsubseteq F^1 \underline{\perp}$:-)

Schluss: Gelte bereits $F^{i-1} \underline{\perp} \sqsubseteq F^i \underline{\perp}$. Dann

$$F^i \underline{\perp} = F\,(F^{i-1} \underline{\perp}) \sqsubseteq F\,(F^i \underline{\perp}) = F^{i+1} \underline{\perp}$$

da F monoton ist :-)

Fazit:

Wenn $\mathbb{D}$ endlich ist, finden wir mit Sicherheit eine Lösung :-)

Fragen:

Fazit:

Wenn $\mathbb{D}$ endlich ist, finden wir mit Sicherheit eine Lösung :-)

Fragen:

1. Gibt es eine kleinste Lösung ?

Fazit:

Wenn $\mathbb{D}$ endlich ist, finden wir mit Sicherheit eine Lösung :-)

Fragen:

1. Gibt es eine kleinste Lösung ?
2. Wenn ja: findet Iteration die kleinste Lösung ??

Fazit:

Wenn $\mathbb{D}$ endlich ist, finden wir mit Sicherheit eine Lösung :-)

Fragen:

1. Gibt es eine kleinste Lösung ?
2. Wenn ja: findet Iteration die kleinste Lösung ??
3. Was, wenn $\mathbb{D}$ nicht endlich ist ???

Satz — Knaster – Tarski

In einem vollständigen Verband $\mathbb{D}$ hat jede monotone Funktion $f : \mathbb{D} \to \mathbb{D}$ einen kleinsten Fixpunkt d_0.

Sei $P = \{d \in \mathbb{D} \mid f\,d \sqsubseteq d\}$ die Menge der Präfixpunkte.

Dann ist $d_0 = \bigsqcap P$.

Bronisław Knaster (1893-1980), topology

Satz Knaster – Tarski

In einem vollständigen Verband $\mathbb{D}$ hat jede monotone Funktion $f : \mathbb{D} \to \mathbb{D}$ einen kleinsten Fixpunkt d_0.

Sei $P = \{d \in \mathbb{D} \mid f\, d \sqsubseteq d\}$ die Menge der Präfixpunkte.

Dann ist $d_0 = \bigsqcap P$.

Beweis:

(1) $d_0 \in P$:

Satz — Knaster – Tarski

In einem vollständigen Verband $\mathbb{D}$ hat jede monotone Funktion $f : \mathbb{D} \to \mathbb{D}$ einen kleinsten Fixpunkt d_0.

Sei $P = \{d \in \mathbb{D} \mid f\, d \sqsubseteq d\}$ die Menge der Präfixpunkte.

Dann ist $d_0 = \bigsqcap P$.

Beweis:

(1) $d_0 \in P$:

$$
\begin{array}{ll}
& f\, d_0 \sqsubseteq f\, d \sqsubseteq d \quad \text{für alle } d \in P \\
\Longrightarrow & f\, d_0 \quad \text{ist untere Schranke von } P \\
\Longrightarrow & f\, d_0 \sqsubseteq d_0 \quad \text{weil } d_0 = \bigsqcap P \\
\Longrightarrow & d_0 \in P \qquad\qquad \text{:-)}
\end{array}
$$

$$f\,d_0 = d_0 : \tag{2}$$

(2) $\quad f\, d_0 = d_0 :$

$$f\, d_0 \sqsubseteq d_0 \quad \text{wegen} \quad (1)$$

$$\Longrightarrow \quad f(f\, d_0) \sqsubseteq f\, d_0 \quad \text{wegen Monotonie von } f$$

$$\Longrightarrow \quad f\, d_0 \in P$$

$$\Longrightarrow \quad d_0 \sqsubseteq f\, d_0 \qquad \text{und die Behauptung folgt} \quad \text{:-)}$$

(2) $\quad f\, d_0 = d_0$:

$$
\begin{array}{ll}
 & f\, d_0 \sqsubseteq d_0 \quad \text{wegen} \quad (1) \\
\Longrightarrow & f(f\, d_0) \sqsubseteq f\, d_0 \quad \text{wegen Monotonie von } f \\
\Longrightarrow & f\, d_0 \in P \\
\Longrightarrow & d_0 \sqsubseteq f\, d_0 \quad \text{und die Behauptung folgt} \quad \text{:-)}
\end{array}
$$

(3) $\quad d_0$ ist kleinster Fixpunkt:

(2) $f\,d_0 = d_0$:

$$
\begin{array}{ll}
 & f\,d_0 \sqsubseteq d_0 \quad \text{wegen} \quad (1) \\
\Longrightarrow & f(f\,d_0) \sqsubseteq f\,d_0 \quad \text{wegen Monotonie von } f \\
\Longrightarrow & f\,d_0 \in P \\
\Longrightarrow & d_0 \sqsubseteq f\,d_0 \qquad \text{und die Behauptung folgt} \quad \text{:-)}
\end{array}
$$

(3) d_0 ist kleinster Fixpunkt:

$$
\begin{array}{ll}
 & f\,d_1 = d_1 \sqsubseteq d_1 \quad \text{weiterer Fixpunkt} \\
\Longrightarrow & d_1 \in P \\
\Longrightarrow & d_0 \sqsubseteq d_1 \qquad \text{:-))}
\end{array}
$$

Bemerkung:

Der kleinste Fixpunkt d_0 ist in P und untere Schranke :-)

$\Longrightarrow$ d_0 ist der kleinste Wert x mit $x \sqsupseteq f\, x$

Bemerkung:

Der kleinste Fixpunkt d_0 ist in P und untere Schranke :-)

$\Longrightarrow$ d_0 ist der kleinste Wert x mit $x \sqsupseteq f\,x$

Anwendung:

Sei $$x_i \sqsupseteq f_i(x_1, \ldots, x_n), \quad i = 1, \ldots, n \qquad (*)$$

ein Ungleichungssystem, wobei alle $f_i : \mathbb{D}^n \to \mathbb{D}$ monoton sind.

Bemerkung:

Der kleinste Fixpunkt d_0 ist in P und untere Schranke :-)

$\Longrightarrow$ d_0 ist der kleinste Wert x mit $x \sqsupseteq f\,x$

Anwendung:

Sei $x_i \sqsupseteq f_i(x_1, \ldots, x_n), \quad i = 1, \ldots, n$ $(*)$

ein Ungleichungssystem, wobei alle $f_i : \mathbb{D}^n \to \mathbb{D}$ monoton sind.

$\Longrightarrow$ kleinste Lösung von $(*)$ $==$ kleinster Fixpunkt von F :-)

Beispiel 1: $\mathbb{D} = 2^U, \quad f\,x = x \cap a \cup b$

Beispiel 1: $\mathbb{D} = 2^U, \quad f\,x = x \cap a \cup b$

f	$f^k\,\bot$	$f^k\,\top$
0	$\emptyset$	$\top$

Beispiel 1: $\mathbb{D} = 2^U, \quad f\,x = x \cap a \cup b$

f	$f^k\,\bot$	$f^k\,\top$
0	$\emptyset$	$\top$
1	b	$a \cup b$

Beispiel 1: $\mathbb{D} = 2^U, \quad f\,x = x \cap a \cup b$

f	$f^k\,\bot$	$f^k\,\top$
0	$\emptyset$	$\top$
1	b	$a \cup b$
2	b	$a \cup b$

Beispiel 1: $\mathbb{D} = 2^U, \quad f\,x = x \cap a \cup b$

f	$f^k\,\perp$	$f^k\,\top$
0	$\emptyset$	$\top$
1	b	$a \cup b$
2	b	$a \cup b$

Beispiel 2: $\mathbb{D} = \mathbb{N} \cup \{\infty\}$

Für die Funktion $\quad f\,x = x + 1 \quad$ ist:

$$f^i\,\perp = f^i\,0 = i \quad \sqsubset \quad i + 1 = f^{i+1}\,\perp$$

Beispiel 1: $\mathbb{D} = 2^U, \quad f\,x = x \cap a \cup b$

f	$f^k\,\bot$	$f^k\,\top$
0	$\emptyset$	$\top$
1	b	$a \cup b$
2	b	$a \cup b$

Beispiel 2: $\mathbb{D} = \mathbb{N} \cup \{\infty\}$

Für die Funktion $f\,x = x + 1$ ist:

$$f^i\,\bot = f^i\,0 = i \quad \sqsubset \quad i + 1 = f^{i+1}\,\bot$$

$\Longrightarrow$ Die normale Iteration erreicht nie einen Fixpunkt :-(

$\Longrightarrow$ Man benötigt manchmal transfinite Iteration :-)

Satz:

Sei $f : \mathbb{D} \to \mathbb{D}$ monoton und $X \subseteq \mathbb{D}$ die kleinste Menge mit:

(a) $\bot \in X$;

(b) $f\,d \in X$ falls $d \in X$;

(c) $\bigsqcup X_0 \in X$ für alle $X_0 \subseteq X$.

// diese Menge existiert offenbar :-)

Satz:

Sei $f : \mathbb{D} \to \mathbb{D}$ monoton und $X \subseteq \mathbb{D}$ die kleinste Menge mit:

(a) $\perp \in X$;

(b) $f\, d \in X$ falls $d \in X$;

(c) $\bigsqcup X_0 \in X$ für alle $X_0 \subseteq X$.

// diese Menge existiert offenbar :-)

Dann ist $d_0 = \bigsqcup X$ der kleinste Fixpunkt von f.

Beweis:

(1) $f\, d_0 \sqsubseteq d_0$ d.h. d_0 ist Präfixpunkt:

Beweis:

(1) $\quad f\, d_0 \sqsubseteq d_0 \quad$ d.h. $\quad d_0 \quad$ ist Präfixpunkt:

$$\begin{array}{lll} & d_0 \in X & \text{wegen (c)} \\ \Longrightarrow & f\, d_0 \in X & \text{wegen (b)} \\ \Longrightarrow & f\, d_0 \sqsubseteq d_0 & \text{:-)} \end{array}$$

Beweis:

(1) $\quad f\, d_0 \sqsubseteq d_0 \quad$ d.h. $\quad d_0 \quad$ ist Präfixpunkt:

$$
\begin{array}{rll}
 & d_0 \in X & \text{wegen (c)} \\
\Longrightarrow & f\, d_0 \in X & \text{wegen (b)} \\
\Longrightarrow & f\, d_0 \sqsubseteq d_0 & \text{:-)}
\end{array}
$$

(2) $\quad d_0 \quad$ ist kleinster Präfixpunkt:

Beweis:

(1) $f\,d_0 \sqsubseteq d_0$ d.h. d_0 ist Präfixpunkt:

$$
\begin{array}{rll}
 & d_0 \in X & \text{wegen (c)} \\
\Longrightarrow & f\,d_0 \in X & \text{wegen (b)} \\
\Longrightarrow & f\,d_0 \sqsubseteq d_0 & \text{:-)}
\end{array}
$$

(2) d_0 ist kleinster Präfixpunkt:

Sei d_1 weiterer Präfixpunkt, d.h. $f\,d_1 \sqsubseteq d_1$.
Dann erfüllt die Menge: $X_1 = \{x \in \mathbb{D} \mid x \sqsubseteq d_1\}$
die Eigenschaften (a), (b) und (c) :-)

Beweis:

(1) $f\, d_0 \sqsubseteq d_0$ d.h. d_0 ist Präfixpunkt:

$$
\begin{array}{rll}
 & d_0 \in X & \text{wegen (c)} \\
\Longrightarrow & f\, d_0 \in X & \text{wegen (b)} \\
\Longrightarrow & f\, d_0 \sqsubseteq d_0 & \text{:-)}
\end{array}
$$

(2) d_0 ist kleinster Präfixpunkt:

Sei d_1 weiterer Präfixpunkt, d.h. $f\, d_1 \sqsubseteq d_1$.
Dann erfüllt die Menge: $X_1 = \{x \in \mathbb{D} \mid x \sqsubseteq d_1\}$
die Eigenschaften (a), (b) und (c) :-)

$$
\begin{array}{rl}
\Longrightarrow & X \subseteq X_1 \\
\Longrightarrow & d_1 \quad \text{ist obere Schranke von} \quad X \\
\Longrightarrow & d_0 = \bigsqcup X \sqsubseteq d_1 \quad \text{:-))}
\end{array}
$$

Fazit:

Wir können Constraint-Systeme durch Fixpunkt-Iteration lösen,
d.h. durch wiederholtes Einsetzen :-)

Fazit:

Wir können Constraint-Systeme durch Fixpunkt-Iteration lösen,
d.h. durch wiederholtes Einsetzen :-)

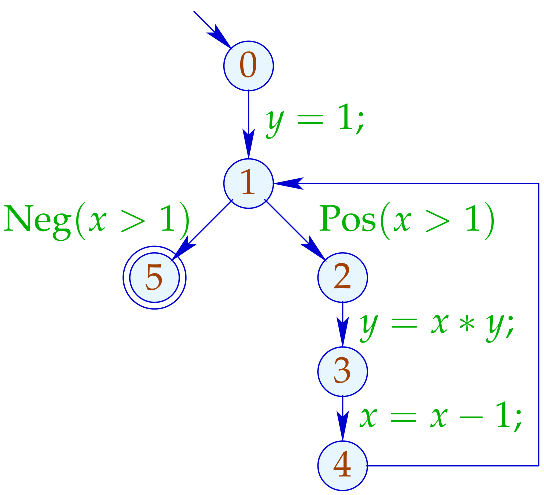
Achtung: Naive Fixpunkt-Iteration ist ziemlich ineffizient :-(

Fazit:

Wir können Constraint-Systeme durch Fixpunkt-Iteration lösen, d.h. durch wiederholtes Einsetzen :-)

Achtung: Naive Fixpunkt-Iteration ist ziemlich ineffizient :-(

Beispiel:



0
1
2
3
4
5

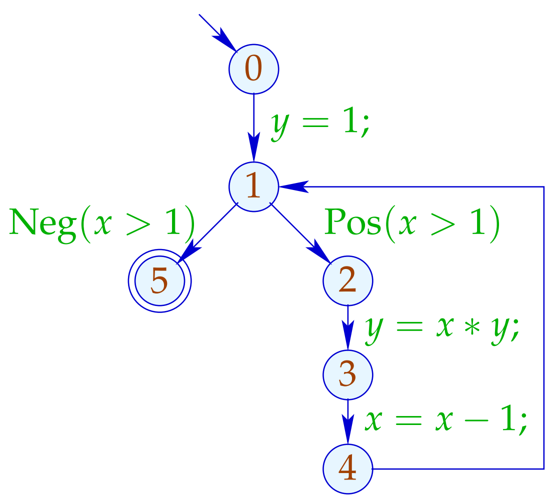
Fazit:

Wir können Constraint-Systeme durch Fixpunkt-Iteration lösen, d.h. durch wiederholtes Einsetzen :-)

Achtung:

Naive Fixpunkt-Iteration ist ziemlich ineffizient :-(

Beispiel:



	1
0	$\emptyset$
1	$\{1, x > 1, x - 1\}$
2	*Expr*
3	$\{1, x > 1, x - 1\}$
4	$\{1\}$
5	*Expr*

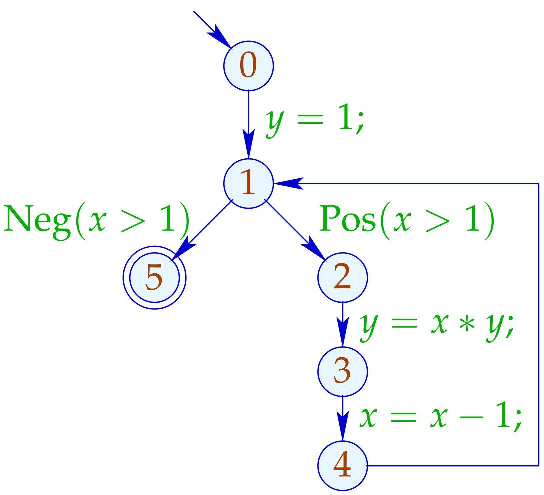
Fazit:

Wir können Constraint-Systeme durch Fixpunkt-Iteration lösen, d.h. durch wiederholtes Einsetzen :-)

Achtung:

Naive Fixpunkt-Iteration ist ziemlich ineffizient :-(

Beispiel:



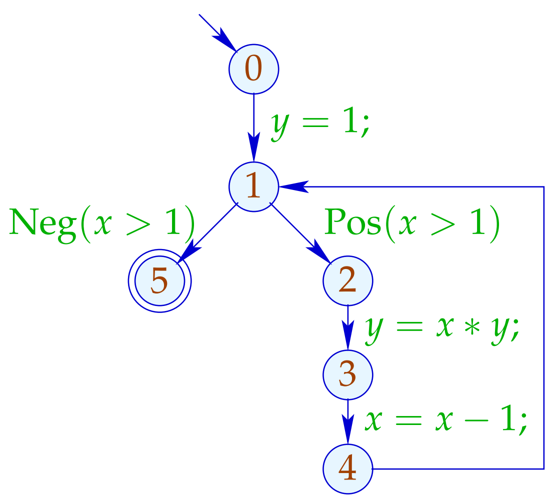
	1	2
0	$\emptyset$	$\emptyset$
1	$\{1, x > 1, x - 1\}$	$\{1\}$
2	*Expr*	$\{1, x > 1, x - 1\}$
3	$\{1, x > 1, x - 1\}$	$\{1, x > 1, x - 1\}$
4	$\{1\}$	$\{1\}$
5	*Expr*	$\{1, x > 1, x - 1\}$

Fazit:

Wir können Constraint-Systeme durch Fixpunkt-Iteration lösen, d.h. durch wiederholtes Einsetzen :-)

Achtung: Naive Fixpunkt-Iteration ist ziemlich ineffizient :-(

Beispiel:



	1	2	3
0	$\emptyset$	$\emptyset$	$\emptyset$
1	$\{1, x > 1, x - 1\}$	$\{1\}$	$\{1\}$
2	*Expr*	$\{1, x > 1, x - 1\}$	$\{1, x > 1\}$
3	$\{1, x > 1, x - 1\}$	$\{1, x > 1, x - 1\}$	$\{1, x > 1, x - 1\}$
4	$\{1\}$	$\{1\}$	$\{1\}$
5	*Expr*	$\{1, x > 1, x - 1\}$	$\{1, x > 1\}$

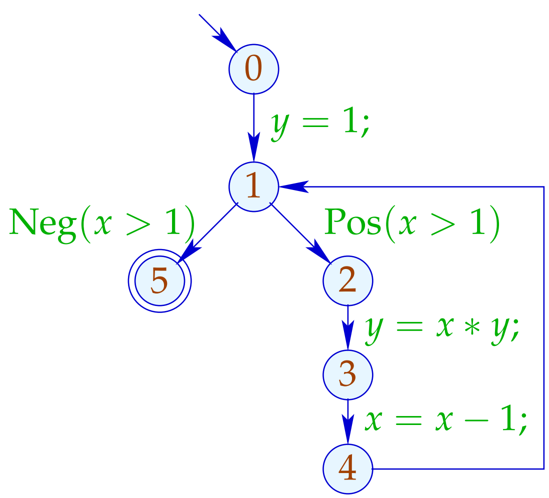
Fazit:

Wir können Constraint-Systeme durch Fixpunkt-Iteration lösen, d.h. durch wiederholtes Einsetzen :-)

Achtung:

Naive Fixpunkt-Iteration ist ziemlich ineffizient :-(

Beispiel:



	1	2	3	4
0	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
1	$\{1, x > 1, x - 1\}$	$\{1\}$	$\{1\}$	$\{1\}$
2	*Expr*	$\{1, x > 1, x - 1\}$	$\{1, x > 1\}$	$\{1, x > 1\}$
3	$\{1, x > 1, x - 1\}$	$\{1, x > 1, x - 1\}$	$\{1, x > 1, x - 1\}$	$\{1, x > 1\}$
4	$\{1\}$	$\{1\}$	$\{1\}$	$\{1\}$
5	*Expr*	$\{1, x > 1, x - 1\}$	$\{1, x > 1\}$	$\{1, x > 1\}$

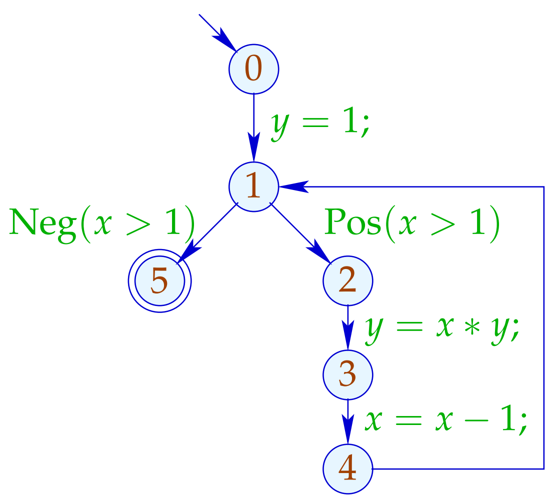
Fazit:

Wir können Constraint-Systeme durch Fixpunkt-Iteration lösen, d.h. durch wiederholtes Einsetzen :-)

Achtung:

Naive Fixpunkt-Iteration ist ziemlich ineffizient :-(

Beispiel:



	1	2	3	4	5
0	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	
1	$\{1, x > 1, x - 1\}$	$\{1\}$	$\{1\}$	$\{1\}$	
2	*Expr*	$\{1, x > 1, x - 1\}$	$\{1, x > 1\}$	$\{1, x > 1\}$	
3	$\{1, x > 1, x - 1\}$	$\{1, x > 1, x - 1\}$	$\{1, x > 1, x - 1\}$	$\{1, x > 1\}$	dito
4	$\{1\}$	$\{1\}$	$\{1\}$	$\{1\}$	
5	*Expr*	$\{1, x > 1, x - 1\}$	$\{1, x > 1\}$	$\{1, x > 1\}$	

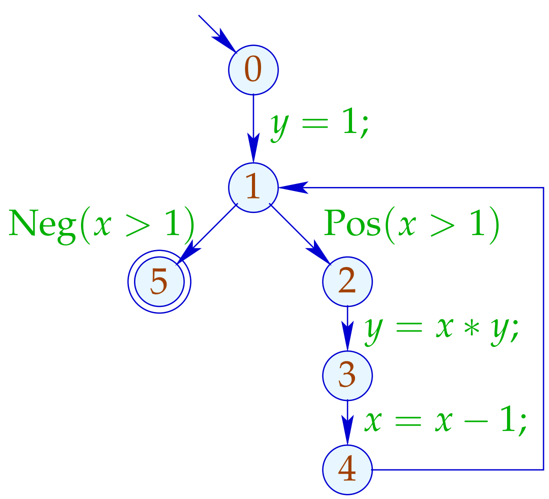
Idee: Round Robin Iteration

Benutze bei der Iteration nicht die Werte der letzten Iteration, sondern die jeweils aktuellen :-)

Idee: Round Robin Iteration

Benutze bei der Iteration nicht die Werte der letzten Iteration, sondern die jeweils aktuellen :-)

Beispiel:

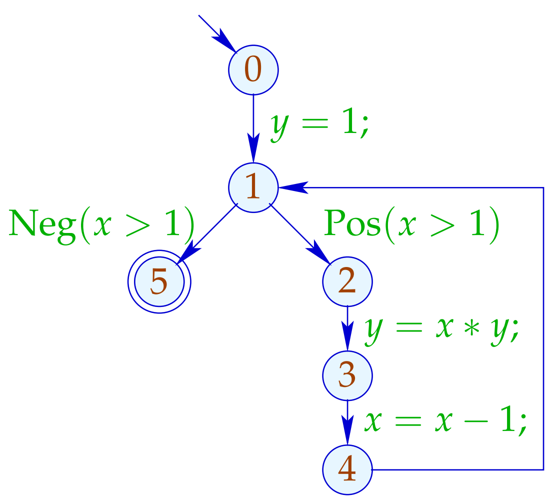


0
1
2
3
4
5

Idee: Round Robin Iteration

Benutze bei der Iteration nicht die Werte der letzten Iteration, sondern die jeweils aktuellen :-)

Beispiel:

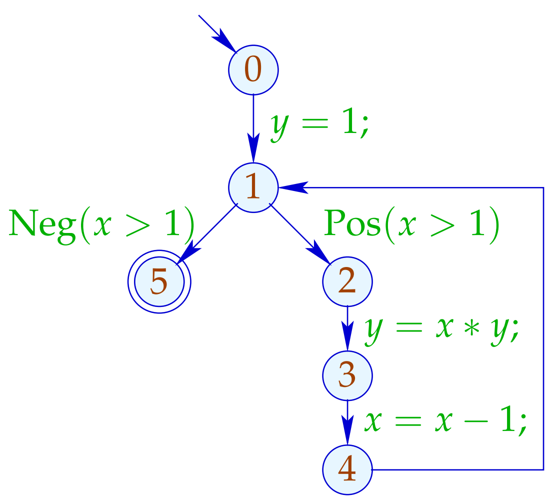


	1
0	$\emptyset$
1	$\{1\}$
2	$\{1, x > 1\}$
3	$\{1, x > 1\}$
4	$\{1\}$
5	$\{1, x > 1\}$

Idee: Round Robin Iteration

Benutze bei der Iteration nicht die Werte der letzten Iteration, sondern die jeweils aktuellen :-)

Beispiel:



	1	2
0	$\emptyset$	
1	$\{1\}$	
2	$\{1, x > 1\}$	
3	$\{1, x > 1\}$	dito
4	$\{1\}$	
5	$\{1, x > 1\}$	

Der Code für Round Robin Iteration sieht in Java so aus:

$$
\begin{array}{l}
\textsf{for}\ (i = 1; i \leq n; i{+}{+})\ x_i = \bot; \\
\textsf{do}\ \{ \\
\qquad \mathit{finished} = \textsf{true}; \\
\qquad \textsf{for}\ (i = 1; i \leq n; i{+}{+})\ \{ \\
\qquad\qquad \mathit{new} = f_i(x_1, \ldots, x_n); \\
\qquad\qquad \textsf{if}\ (!(x_i \sqsupseteq \mathit{new}))\ \{ \\
\qquad\qquad\qquad \mathit{finished} = \textsf{false}; \\
\qquad\qquad\qquad x_i = x_i \sqcup \mathit{new}; \\
\qquad\qquad \} \\
\qquad \} \\
\}\ \textsf{while}\ (!\mathit{finished});
\end{array}
$$

Zur Korrektheit:

Sei $y_i^{(d)}$ die i-te Komponente von $F^d\,\underline{\perp}$.

Sei $x_i^{(d)}$ der Wert von x_i nach der i-ten RR-Iteration.

Zur Korrektheit:

Sei $y_i^{(d)}$ die i-te Komponente von $F^d \underline{\perp}$.

Sei $x_i^{(d)}$ der Wert von x_i nach der i-ten RR-Iteration.

Man zeigt:

(1) $y_i^{(d)} \sqsubseteq x_i^{(d)}$:-)

Zur Korrektheit:

Sei $y_i^{(d)}$ die *i*-te Komponente von $F^d \underline{\perp}$.

Sei $x_i^{(d)}$ der Wert von x_i nach der *i*-ten RR-Iteration.

Man zeigt:

(1) $y_i^{(d)} \sqsubseteq x_i^{(d)}$:-)

(2) $x_i^{(d)} \sqsubseteq z_i$ für jede Lösung $(z_1, \ldots, z_n)$:-)

Zur Korrektheit:

Sei $y_i^{(d)}$ die i-te Komponente von $F^d\,\underline{\perp}$.

Sei $x_i^{(d)}$ der Wert von x_i nach der i-ten RR-Iteration.

Man zeigt:

(1) $y_i^{(d)} \sqsubseteq x_i^{(d)}$:-)

(2) $x_i^{(d)} \sqsubseteq z_i$ für jede Lösung $(z_1, \ldots, z_n)$:-)

(3) Terminiert RR-Iteration nach d Runden, ist $(x_1^{(d)}, \ldots, x_n^{(d)})$ eine Lösung :-))

Achtung:

Die Effizienz von RR-Iteration hängt von der Anordnung der Variablen ab !!!

Achtung:

Die Effizienz von RR-Iteration hängt von der Anordnung der Variablen ab !!!

Günstig:

$\longrightarrow$ u vor v, falls $u \longrightarrow^* v$;

$\longrightarrow$ Eintrittsbedingung vor Schleifen-Rumpf :-)

Achtung:

Die Effizienz von RR-Iteration hängt von der Anordnung der Variablen ab !!!

Günstig:

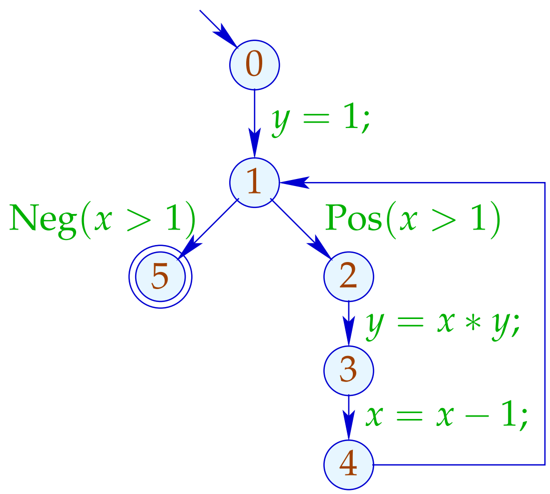
$\rightarrow$ u vor v, falls $u \rightarrow^* v$;

$\rightarrow$ Eintrittsbedingung vor Schleifen-Rumpf :-)

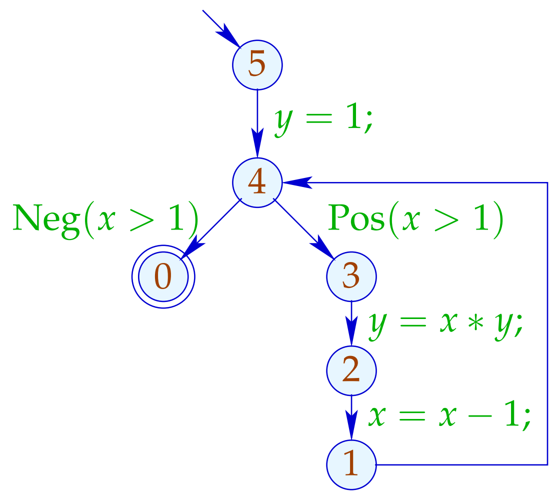
Ungünstig:

z.B. post-order DFS auf dem CFG, startend von start :-)

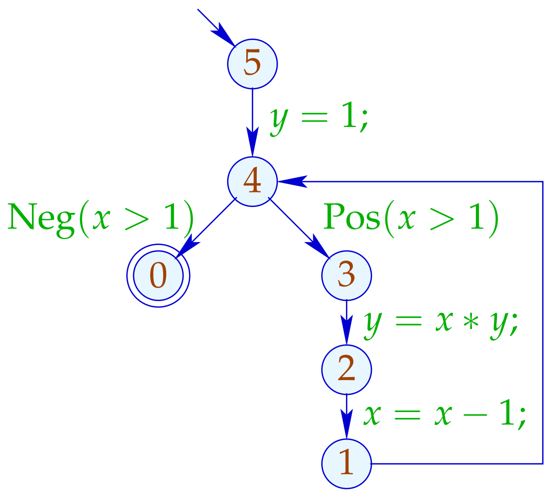
Günstig:



Ungünstig:

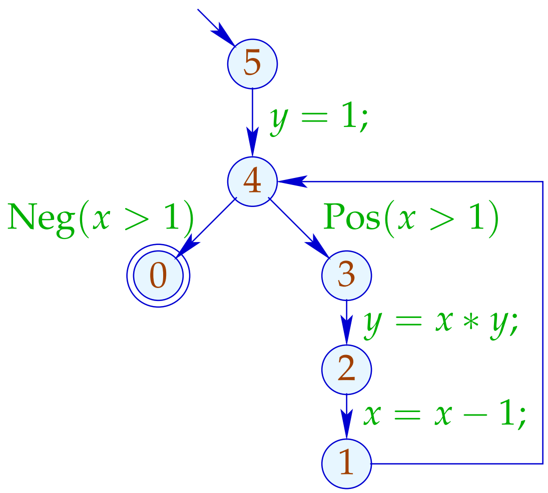


Ungünstige Round Robin Iteration:



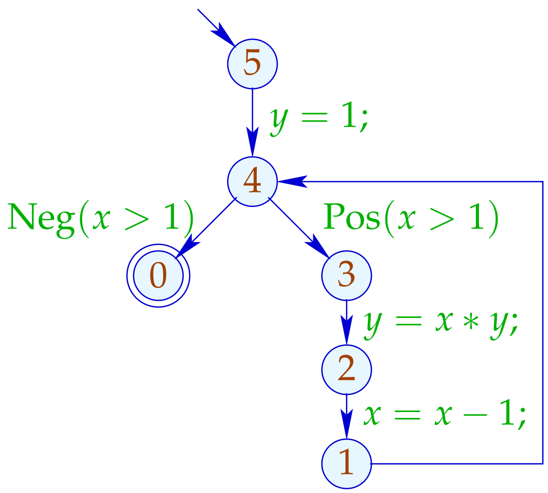
0
1
2
3
4
5

Ungünstige Round Robin Iteration:



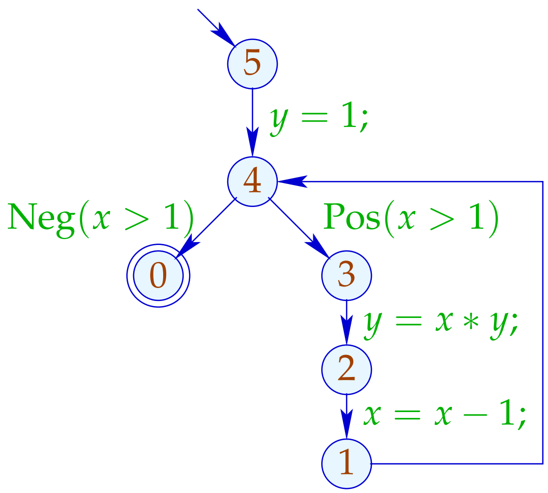
	1
0	*Expr*
1	$\{1\}$
2	$\{1, x-1, x>1\}$
3	*Expr*
4	$\{1\}$
5	$\emptyset$

Ungünstige Round Robin Iteration:



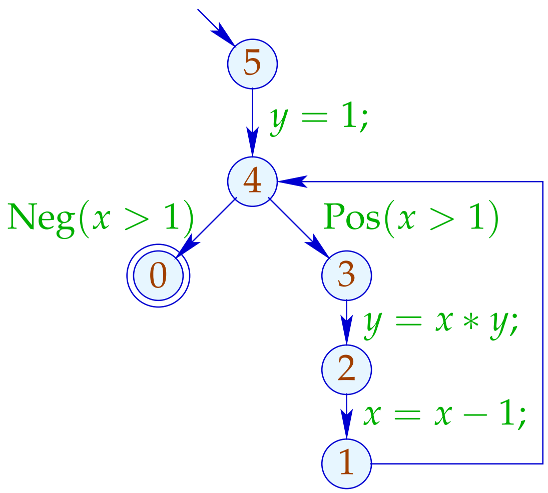
	1	2
0	*Expr*	$\{1, x > 1\}$
1	$\{1\}$	$\{1\}$
2	$\{1, x - 1, x > 1\}$	$\{1, x - 1, x > 1\}$
3	*Expr*	$\{1, x > 1\}$
4	$\{1\}$	$\{1\}$
5	$\emptyset$	$\emptyset$

Ungünstige Round Robin Iteration:



	1	2	3
0	*Expr*	$\{1, x > 1\}$	$\{1, x > 1\}$
1	$\{1\}$	$\{1\}$	$\{1\}$
2	$\{1, x-1, x > 1\}$	$\{1, x-1, x > 1\}$	$\{1, x > 1\}$
3	*Expr*	$\{1, x > 1\}$	$\{1, x > 1\}$
4	$\{1\}$	$\{1\}$	$\{1\}$
5	$\emptyset$	$\emptyset$	$\emptyset$

Ungünstige Round Robin Iteration:



	1	2	3	4
0	*Expr*	$\{1, x > 1\}$	$\{1, x > 1\}$	
1	$\{1\}$	$\{1\}$	$\{1\}$	
2	$\{1, x - 1, x > 1\}$	$\{1, x - 1, x > 1\}$	$\{1, x > 1\}$	dito
3	*Expr*	$\{1, x > 1\}$	$\{1, x > 1\}$	
4	$\{1\}$	$\{1\}$	$\{1\}$	
5	$\emptyset$	$\emptyset$	$\emptyset$	

$\Longrightarrow$ deutlich weniger effizient :-)

... Ende des Exkurses: Vollständige Verbände

... Ende des Exkurses: Vollständige Verbände

Letzte Frage:

Wieso hilft uns eine (oder die kleinste) Lösung des Constraint-Systems weiter ???

... Ende des Exkurses: Vollständige Verbände

Letzte Frage:

Wieso hilft uns eine (oder die kleinste) Lösung des Constraint-Systems weiter ???

Betrachte für einen vollständigen Verband $\mathbb{D}$ Systeme:

$$\begin{array}{lll} \mathcal{I}[\mathit{start}] & \sqsupseteq & d_0 \\ \mathcal{I}[v] & \sqsupseteq & [\![k]\!]^\sharp\,(\mathcal{I}[u]) \qquad k = (u, \_, v) \quad \text{Kante} \end{array}$$

wobei $d_0 \in \mathbb{D}$ und alle $[\![k]\!]^\sharp : \mathbb{D} \to \mathbb{D}$ monoton sind ...

... Ende des Exkurses: Vollständige Verbände

Letzte Frage:

Wieso hilft uns eine (oder die kleinste) Lösung des Constraint-Systems weiter ???

Betrachte für einen vollständigen Verband $\mathbb{D}$ Systeme:

$$\begin{array}{lll} \mathcal{I}[start] & \sqsupseteq & d_0 \\ \mathcal{I}[v] & \sqsupseteq & [\![k]\!]^\sharp\,(\mathcal{I}[u]) \qquad k = (u, \_, v) \quad \text{Kante} \end{array}$$

wobei $d_0 \in \mathbb{D}$ und alle $[\![k]\!]^\sharp : \mathbb{D} \to \mathbb{D}$ monoton sind ...

$\Longrightarrow$ monotoner Analyse-Rahmen

Gesucht: MOP (Merge Over all Paths)

$$\mathcal{I}^*[v] = \bigsqcup\{[\![\pi]\!]^\sharp\, d_0 \mid \pi : start \rightarrow^* v\}$$

Gesucht: MOP (Merge Over all Paths)

$$\mathcal{I}^*[v] = \bigsqcup \{ [\![\pi]\!]^\sharp\, d_0 \mid \pi : \mathit{start} \rightarrow^* v \}$$

Theorem Kam, Ullman 1975

Sei $\mathcal{I}$ die kleinste Lösung des Constraint-Systems. Dann gilt:

$$\mathcal{I}[v] \sqsupseteq \mathcal{I}^*[v] \qquad \text{für jedes} \quad v$$

Jeffrey D. Ullman, Stanford

Gesucht: **MOP** (Merge Over all Paths)

$$\mathcal{I}^*[v] = \bigsqcup \{ [\![\pi]\!]^\sharp \, d_0 \mid \pi : \mathit{start} \to^* v \}$$

Theorem Kam, Ullman 1975

Sei $\mathcal{I}$ die kleinste Lösung des Constraint-Systems. Dann gilt:

$$\mathcal{I}[v] \sqsupseteq \mathcal{I}^*[v] \qquad \text{für jedes} \quad v$$

Insbesondere: $\mathcal{I}[v] \sqsupseteq [\![\pi]\!]^\sharp \, d_0$ für jedes $\pi : \mathit{start} \to^* v$

Beweis: Induktion nach der Länge von π.

Beweis: Induktion nach der Länge von π.

Anfang: $\pi = \epsilon$ (leerer Pfad)

Beweis: Induktion nach der Länge von π.

Anfang: $\pi = \epsilon$ (leerer Pfad)

Dann gilt:

$$[\![\pi]\!]^\sharp\, d_0 = [\![\epsilon]\!]^\sharp\, d_0 = d_0 \sqsubseteq \mathcal{I}[\mathit{start}]$$

Beweis: Induktion nach der Länge von π.

Anfang: $\pi = \epsilon$ (leerer Pfad)

Dann gilt:

$$[\![\pi]\!]^\sharp\, d_0 = [\![\epsilon]\!]^\sharp\, d_0 = d_0 \sqsubseteq \mathcal{I}[start]$$

Schluss: $\pi = \pi' k$ für $k = (u, \_, v)$ Kante.

Beweis: Induktion nach der Länge von π.

Anfang: $\pi = \epsilon$ (leerer Pfad)

Dann gilt:

$$[\![\pi]\!]^\sharp\, d_0 = [\![\epsilon]\!]^\sharp\, d_0 = d_0 \sqsubseteq \mathcal{I}[start]$$

Schluss: $\pi = \pi' k$ für $k = (u, \_, v)$ Kante.

Dann gilt:

$$\begin{array}{lllll} & [\![\pi']\!]^\sharp\, d_0 & \sqsubseteq & \mathcal{I}[u] & \text{wegen I.H. für } \pi \\ \Longrightarrow & [\![\pi]\!]^\sharp\, d_0 & = & [\![k]\!]^\sharp\, ([\![\pi']\!]^\sharp\, d_0) & \\ & & \sqsubseteq & [\![k]\!]^\sharp\, (\mathcal{I}[u]) & \text{da } [\![k]\!]^\sharp \text{ monoton} \\ & & \sqsubseteq & \mathcal{I}[v] & \text{da } \mathcal{I} \text{ Lösung :-))} \end{array}$$

Enttäuschung:

Liefern Lösungen des Constraint-Systems nur obere Schranken ???

Enttäuschung:

Liefern Lösungen des Constraint-Systems nur obere Schranken ???

Antwort:

Im allgemeinen: ja   :-(

Enttäuschung:

Liefern Lösungen des Constraint-Systems nur obere Schranken ???

Antwort:

Im allgemeinen: ja :-(

Es sei denn, alle Funktionen $[\![k]\!]^\sharp$ sind distributiv ... :-)

Die Funktion $f : \mathbb{D}_1 \longrightarrow \mathbb{D}_2$ heißt

- distributiv, falls $f\,(\bigsqcup X) = \bigsqcup\{f\,x \mid x \in X\}$ für alle $\emptyset \neq X \subseteq \mathbb{D}$;
- strikt, falls $f\,\bot = \bot$.
- total distributiv, falls f distributiv und strikt ist.

Die Funktion $f : \mathbb{D}_1 \to \mathbb{D}_2$ heißt

- distributiv, falls $f\,(\bigsqcup X) = \bigsqcup\{f\,x \mid x \in X\}$ für alle $\emptyset \neq X \subseteq \mathbb{D}$;
- strikt, falls $f\,\perp = \perp$.
- total distributiv, falls f distributiv und strikt ist.

Beispiele:

- $f\,x = x \cap a \cup b$ für $a, b \subseteq U$.

Die Funktion $f : \mathbb{D}_1 \to \mathbb{D}_2$ heißt

- distributiv, falls $f\,(\bigsqcup X) = \bigsqcup\{f\,x \mid x \in X\}$ für alle $\emptyset \neq X \subseteq \mathbb{D}$;
- strikt, falls $f\,\bot = \bot$.
- total distributiv, falls f distributiv und strikt ist.

Beispiele:

- $f\,x = x \cap a \cup b$ für $a, b \subseteq U$.

  Striktheit: $f\,\emptyset = a \cap \emptyset \cup b = b \quad = \quad \emptyset$ sofern $b = \emptyset$:-(

Die Funktion $f : \mathbb{D}_1 \to \mathbb{D}_2$ heißt

- distributiv, falls $f(\bigsqcup X) = \bigsqcup\{f\,x \mid x \in X\}$ für alle $\emptyset \neq X \subseteq \mathbb{D}$;
- strikt, falls $f\,\bot = \bot$.
- total distributiv, falls f distributiv und strikt ist.

Beispiele:

- $f\,x = x \cap a \cup b$ für $a, b \subseteq U$.

  Striktheit: $f\,\emptyset = a \cap \emptyset \cup b = b = \emptyset$ sofern $b = \emptyset$:-(

  Distributivität:

$$
\begin{array}{lll}
f\,(x_1 \cup x_2) & = & a \cap (x_1 \cup x_2) \cup b \\
 & = & a \cap x_1 \cup a \cap x_2 \cup b \\
 & = & f\,x_1 \cup f\,x_2 \qquad\qquad \text{:-)}
\end{array}
$$

- $\mathbb{D}_1 = \mathbb{D}_2 = \mathbb{N} \cup \{\infty\}$, $\quad \mathsf{inc}\, x = x + 1$

- $\mathbb{D}_1 = \mathbb{D}_2 = \mathbb{N} \cup \{\infty\}$, $\quad \mathsf{inc}\, x = x + 1$

  Striktheit: $\quad f\,\perp = \mathsf{inc}\, 0 = 1 \quad \neq \quad \perp \quad$:-(

- $\mathbb{D}_1 = \mathbb{D}_2 = \mathbb{N} \cup \{\infty\}, \quad \mathsf{inc}\, x = x + 1$

  Striktheit: $\quad f\,\bot = \mathsf{inc}\, 0 = 1 \quad \neq \quad \bot$:-(

  Distributivität: $\quad f\,(\bigsqcup X) \quad = \quad \bigsqcup\{x + 1 \mid x \in X\}$ für $\emptyset \neq X$:-)

- $\mathbb{D}_1 = \mathbb{D}_2 = \mathbb{N} \cup \{\infty\}, \quad \mathsf{inc}\, x = x + 1$

  Striktheit: $\quad f\,\bot = \mathsf{inc}\, 0 = 1 \quad \neq \quad \bot \quad$:-(

  Distributivität: $\quad f\,(\bigsqcup X) \quad = \quad \bigsqcup\{x + 1 \mid x \in X\}$ für $\emptyset \neq X$:-)

- $\mathbb{D}_1 = (\mathbb{N} \cup \{\infty\})^2, \quad \mathbb{D}_2 = \mathbb{N} \cup \{\infty\}, \quad f(x_1, x_2) = x_1 + x_2$

- $\mathbb{D}_1 = \mathbb{D}_2 = \mathbb{N} \cup \{\infty\}, \quad \mathsf{inc}\, x = x + 1$

  Striktheit: $f\,\bot = \mathsf{inc}\, 0 = 1 \quad \neq \quad \bot \quad$:-(

  Distributivität: $f\,(\bigsqcup X) \quad = \quad \bigsqcup\{x + 1 \mid x \in X\}$ für $\emptyset \neq X$:-)

- $\mathbb{D}_1 = (\mathbb{N} \cup \{\infty\})^2, \quad \mathbb{D}_2 = \mathbb{N} \cup \{\infty\}, \quad f(x_1, x_2) = x_1 + x_2:$

  Striktheit: $f\,\bot = 0 + 0 \quad = \quad 0$:-)

- $\mathbb{D}_1 = \mathbb{D}_2 = \mathbb{N} \cup \{\infty\}, \quad \mathsf{inc}\, x = x + 1$

  Striktheit: $f\,\bot = \mathsf{inc}\, 0 = 1 \quad \neq \quad \bot$:-(

  Distributivität: $f\,(\bigsqcup X) \quad = \quad \bigsqcup\{x + 1 \mid x \in X\}$ für $\emptyset \neq X$:-)

- $\mathbb{D}_1 = (\mathbb{N} \cup \{\infty\})^2, \quad \mathbb{D}_2 = \mathbb{N} \cup \{\infty\}, \quad f(x_1, x_2) = x_1 + x_2 :$

  Striktheit: $f\,\bot = 0 + 0 \quad = \quad 0$:-)

  Distributivität:

$$\begin{array}{lllll} f\,((1,4) \sqcup (4,1)) & = & f\,(4,4) & = & 8 \\ & \neq & 5 & = & f\,(1,4) \sqcup f\,(4,1) \end{array}$$:-)

Bemerkung:

Ist $f : \mathbb{D}_1 \to \mathbb{D}_2$ distributiv, dann auch monoton :-)

Bemerkung:

Ist $f : \mathbb{D}_1 \to \mathbb{D}_2$ distributiv, dann auch monoton :-)

Offenbar gilt: $a \sqsubseteq b$ gdw. $a \sqcup b = b$.

Bemerkung:

Ist $f : \mathbb{D}_1 \to \mathbb{D}_2$ distributiv, dann auch monoton :-)

Offenbar gilt: $a \sqsubseteq b$ gdw. $a \sqcup b = b$.

Daraus folgt:

$$\begin{array}{rrcl} & f\,b & = & f\,(a \sqcup b) \\ & & = & f\,a \sqcup f\,b \\ \Longrightarrow & f\,a & \sqsubseteq & f\,b \qquad\qquad \text{:-)} \end{array}$$

Annahme: alle v sind von *start* erreichbar.

Annahme: alle v sind von *start* erreichbar.
Dann gilt:

Theorem Kildall 1972

Sind alle Kanten-Effekte $[\![k]\!]^\sharp$ distributiv, dann ist:
$\mathcal{I}^*[v] = \mathcal{I}[v]$ für alle v .

Gary A. Kildall (1942-1994).

Hat später am Betriebssystem CP/M und an GUIs für PCs gearbeitet.

Annahme: alle v sind von *start* erreichbar.
Dann gilt:

Theorem

Kildall 1972

Sind alle Kanten-Effekte $[\![k]\!]^\sharp$ distributiv, dann ist:
$\mathcal{I}^*[v] = \mathcal{I}[v]$ für alle v .

Annahme: alle v sind von *start* erreichbar.
Dann gilt:

Theorem

Kildall 1972

Sind alle Kanten-Effekte $[\![k]\!]^\sharp$ distributiv, dann ist:
$\mathcal{I}^*[v] = \mathcal{I}[v]$ für alle v .

Beweis:

Offenbar genügt es zu zeigen, dass $\mathcal{I}^*$ eine Lösung ist :-)

Wir zeigen, dass $\mathcal{I}^*$ alle Ungleichungen erfüllt :-))

Wir zeigen, dass $\mathcal{I}^*$ alle Ungleichungen erfüllt :-))

(1) Für *start* zeigen wir:

$$\begin{array}{lll} \mathcal{I}^*[\mathit{start}] & = & \bigsqcup\{[\![\pi]\!]^\sharp\, d_0 \mid \pi : \mathit{start} \to^* \mathit{start}\} \\ & \sqsupseteq & [\![\epsilon]\!]^\sharp\, d_0 \\ & \sqsupseteq & d_0 \qquad \text{:-)} \end{array}$$

Wir zeigen, dass $\mathcal{I}^*$ alle Ungleichungen erfüllt :-))

(1) Für *start* zeigen wir:

$$\begin{array}{lll} \mathcal{I}^*[start] & = & \bigsqcup\{[\![\pi]\!]^\sharp\, d_0 \mid \pi : start \to^* start\} \\ & \sqsupseteq & [\![\epsilon]\!]^\sharp\, d_0 \\ & \sqsupseteq & d_0 \qquad :\text{-)} \end{array}$$

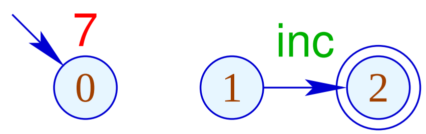
(2) Für jedes $k = (u, \_, v)$ zeigen wir:

$$\begin{array}{lll} \mathcal{I}^*[v] & = & \bigsqcup\{[\![\pi]\!]^\sharp\, d_0 \mid \pi : start \to^* v\} \\ & \sqsupseteq & \bigsqcup\{[\![\pi' k]\!]^\sharp\, d_0 \mid \pi' : start \to^* u\} \\ & = & \bigsqcup\{[\![k]\!]^\sharp\, ([\![\pi']\!]^\sharp\, d_0) \mid \pi' : start \to^* u\} \\ & = & [\![k]\!]^\sharp\, (\bigsqcup\{[\![\pi']\!]^\sharp\, d_0 \mid \pi' : start \to^* u\}) \\ & = & [\![k]\!]^\sharp\, (\mathcal{I}^*[u]) \end{array}$$

da $\{\pi' \mid \pi' : start \to^* u\}$ nicht-leer ist :-)

Achtung:

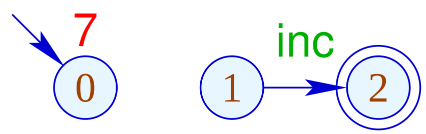
- Auf die Erreichbarkeit aller Programm-Punkt können wir nicht verzichten. Betrachte:



wobei $\mathbb{D} = \mathbb{N} \cup \{\infty\}$

Achtung:

- Auf die Erreichbarkeit aller Programm-Punkt können wir nicht verzichten. Betrachte:



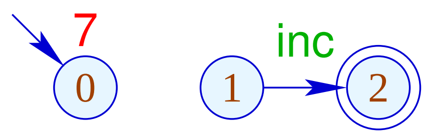
wobei $\mathbb{D} = \mathbb{N} \cup \{\infty\}$

Dann ist:

$$
\begin{array}{lllll}
\mathcal{I}[2] & = & \mathsf{inc}\,0 & = & 1 \\
\mathcal{I}^*[2] & = & \bigsqcup \emptyset & = & 0
\end{array}
$$

Achtung:

- Auf die Erreichbarkeit aller Programm-Punkt können wir nicht verzichten. Betrachte:



wobei $\mathbb{D} = \mathbb{N} \cup \{\infty\}$

Dann ist:

$$\begin{array}{lllll} \mathcal{I}[2] & = & \mathsf{inc}\,0 & = & 1 \\ \mathcal{I}^*[2] & = & \bigsqcup \emptyset & = & 0 \end{array}$$

- Unerreichbare Programmpunkte können wir aber stets wegwerfen :-)

Zusammenfassung und Anwendung:

$\longrightarrow$ Die Kanteneffekte der Analyse zur Verfügbarkeit von Ausdrücken sind distributiv:

$$\begin{aligned} (a \cup (x_1 \cap x_2))\backslash b &= ((a \cup x_1) \cap (a \cup x_2))\backslash b \\ &= ((a \cup x_1)\backslash b) \cap ((a \cup x_2)\backslash b) \end{aligned}$$

Zusammenfassung und Anwendung:

→ Die Kanteneffekte der Analyse zur Verfügbarkeit von Ausdrücken sind distributiv:

$$\begin{aligned}(a \cup (x_1 \cap x_2)) \backslash b &= ((a \cup x_1) \cap (a \cup x_2)) \backslash b \\ &= ((a \cup x_1) \backslash b) \cap ((a \cup x_2) \backslash b)\end{aligned}$$

→ Sind alle Kanteneffekte distributiv, lässt sich der MOP mithilfe des Constraint-Systems und RR-Iteration ausrechnen :-)

Zusammenfassung und Anwendung:

$\longrightarrow$ Die Kanteneffekte der Analyse zur Verfügbarkeit von Ausdrücken sind distributiv:

$$\begin{array}{lll} (a \cup (x_1 \cap x_2)) \backslash b & = & ((a \cup x_1) \cap (a \cup x_2)) \backslash b \\ & = & ((a \cup x_1) \backslash b) \cap ((a \cup x_2) \backslash b) \end{array}$$

$\longrightarrow$ Sind alle Kanteneffekte distributiv, lässt sich der MOP mithilfe des Constraint-Systems und RR-Iteration ausrechnen :-)

$\longrightarrow$ Sind nicht alle Kanteneffekte distributiv, lässt sich eine sichere obere Schranke für den MOP mithilfe des Constraint-Systems und RR-Iteration berechnen :-)

1.2 Beseitigung überflüssiger Zuweisungen

Beispiel:

$$\begin{array}{ll} 1: & x = y + 2; \\ 2: & y = 5; \\ 3: & x = y + 3; \end{array}$$

Der Wert von x an den Programmpunkten $1, 2$ wird überschrieben, bevor er benutzt werden kann.

Die Variable x nennen wir deshalb an diesen Programmpunkten tot :-)

Beachte:

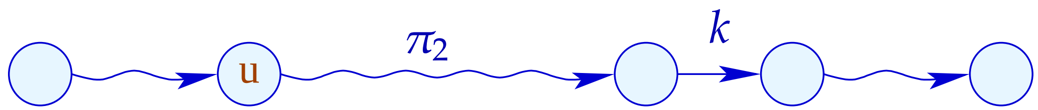
→ Zuweisungen an tote Variablen können wir uns schenken ;-)

→ Solche Ineffizienzen können u.a. durch andere Transformationen hervorgerufen werden.

Formale Definition:

Die Variable x heißt lebendig an u entlang des Pfads π, falls sich π zerlegen lässt in $\pi = \pi_1\, \pi_2\, k\, \pi_3$ so dass gilt:

- π_1 erreicht u ;
- k ist eine Benutzung von x ;
- π_2 enthält keine Überschreibung von x.

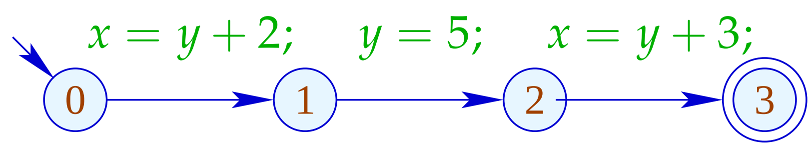


Die Menge der an einer Kante $k = (\_, lab, \_)$ benutzten bzw. überschriebenen Variablen ist dabei gegeben durch:

lab	benutzt	überschrieben
;	$\emptyset$	$\emptyset$
Pos (e)	$Vars\,(e)$	$\emptyset$
Neg (e)	$Vars\,(e)$	$\emptyset$
$R = e;$	$Vars\,(e)$	$\{R\}$
$R_1 = M[R_2];$	$\{R_2\}$	$\{R_1\}$
$M[R_1] = R_2;$	$\{R_1, R_2\}$	$\emptyset$

Eine Variable x , die nicht lebendig an u entlang π ist, heißt tot an u entlang π.

Beispiel:



Wir bemerken:

	lebendig	tot
0	$\{y\}$	$\{x\}$
1	$\emptyset$	$\{x, y\}$
2	$\{y\}$	$\{x\}$
3	$\emptyset$	$\{x, y\}$

Die Variable x ist lebendig an u falls x lebendig ist an u entlang irgend eines Pfads. Andernfalls ist x tot an u .

Die Variable x ist lebendig an u falls x lebendig ist an u entlang irgend eines Pfads. Andernfalls ist x tot an u .

Frage:

Wie berechnet man für jedes u die Menge der dort lebendigen/toten Variablen ???

Die Variable x ist lebendig an u falls x lebendig ist an u entlang irgend eines Pfads. Andernfalls ist x tot an u .

Frage:

Wie berechnet man für jedes u die Menge der dort lebendigen/toten Variablen ???

Idee:

Definiere für jede Kante $k = (u, \_, v)$ eine Funktion $[\![k]\!]^\sharp$, die die Menge der an v lebendigen Variablen in die Menge der an u lebendigen Variablen transformiert ...

Sei $\mathbb{L} = 2^{Vars}$.

Für $k = (\_, lab, \_)$ definieren wir $[\![k]\!]^\sharp = [\![lab]\!]^\sharp$ durch:

$$
\begin{array}{lll}
[\![;]\!]^\sharp\, L & = & L \\
[\![\mathrm{Pos}(e)]\!]^\sharp\, L & = & [\![\mathrm{Neg}(e)]\!]^\sharp\, L \quad = \quad L \cup Vars(e) \\
[\![x = e;]\!]^\sharp\, L & = & (L \backslash \{x\}) \cup Vars(e) \\
[\![R_1 = M[R_2];]\!]^\sharp\, L & = & (L \backslash \{R_1\}) \cup \{R_2\} \\
[\![M[R_1] = R_2;]\!]^\sharp\, L & = & L \cup \{R_1, R_2\}
\end{array}
$$

Sei $\mathbb{L} = 2^{Vars}$.

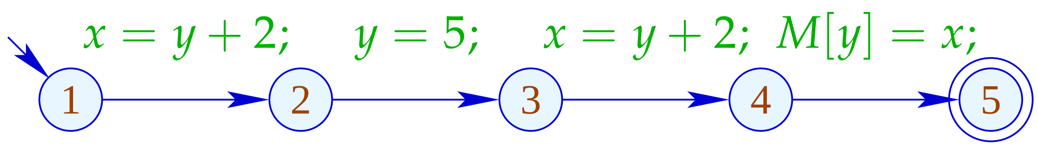
Für $k = (\_, lab, \_)$ definieren wir $[\![k]\!]^\sharp = [\![lab]\!]^\sharp$ durch:

$$
\begin{array}{lll}
[\![;]\!]^\sharp\, L & = & L \\
[\![\mathrm{Pos}(e)]\!]^\sharp\, L & = & [\![\mathrm{Neg}(e)]\!]^\sharp\, L \quad = \quad L \cup Vars(e) \\
[\![x = e;]\!]^\sharp\, L & = & (L \backslash \{x\}) \cup Vars(e) \\
[\![R_1 = M[R_2];]\!]^\sharp\, L & = & (L \backslash \{R_1\}) \cup \{R_2\} \\
[\![M[R_1] = R_2;]\!]^\sharp\, L & = & L \cup \{R_1, R_2\}
\end{array}
$$

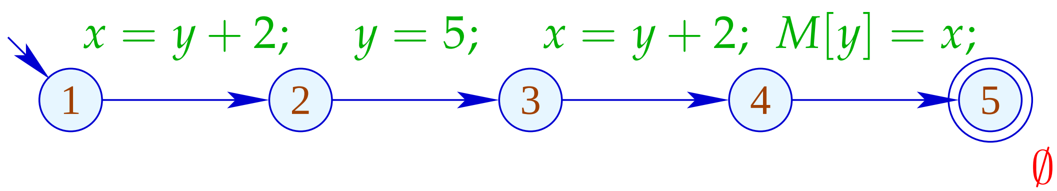
$[\![k]\!]^\sharp$ können wir wieder zu Effekten $[\![\pi]\!]^\sharp$ ganzer Pfade $\pi = k_1 \ldots k_r$ fortsetzen durch:

$$[\![\pi]\!]^\sharp = [\![k_1]\!]^\sharp \circ \ldots \circ [\![k_r]\!]^\sharp$$

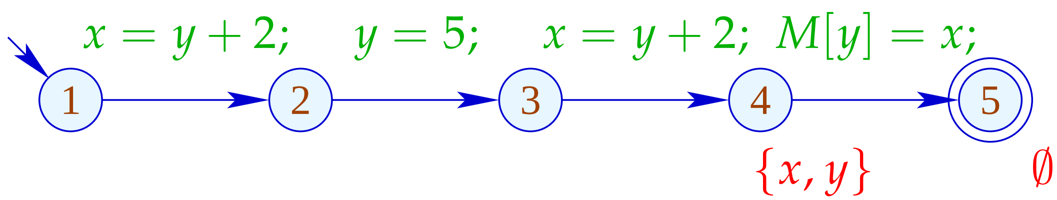
Wir vergewissern uns, dass diese Definitionen vernünftig sind :-)



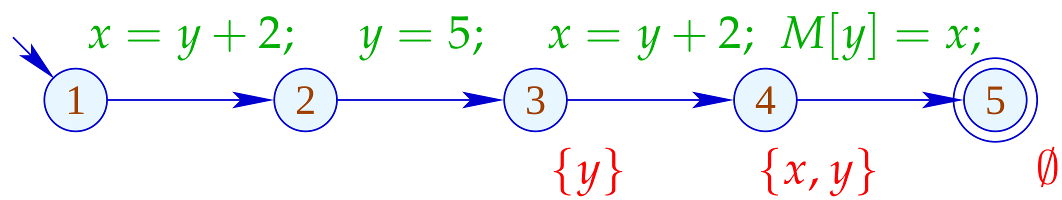
Wir vergewissern uns, dass diese Definitionen vernünftig sind :-)



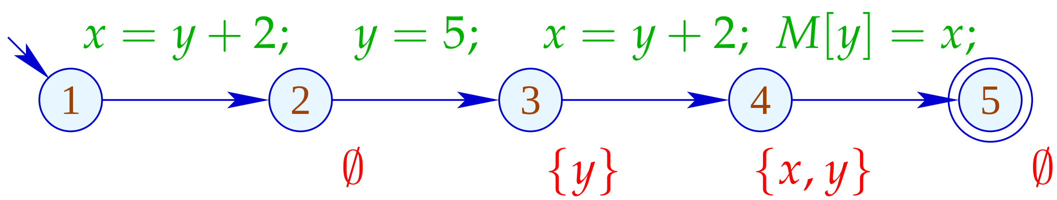
Wir vergewissern uns, dass diese Definitionen vernünftig sind :-)



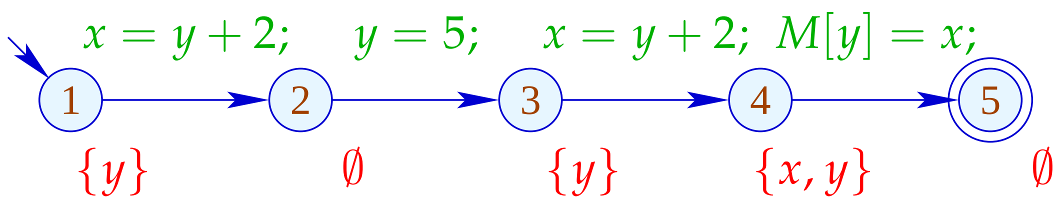
Wir vergewissern uns, dass diese Definitionen vernünftig sind :-)



Wir vergewissern uns, dass diese Definitionen vernünftig sind :-)



Wir vergewissern uns, dass diese Definitionen vernünftig sind :-)



Die Menge der an u lebendigen Variablen ist dann:

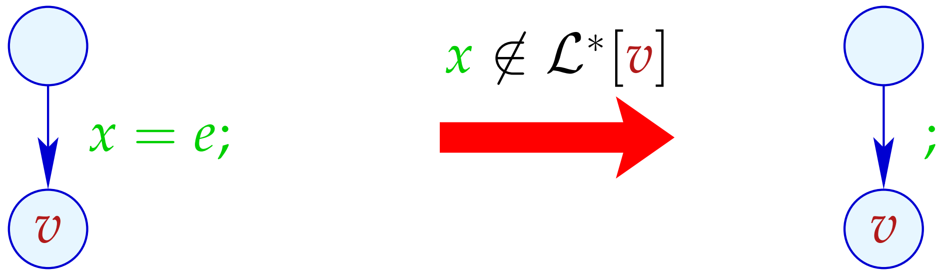
$$\mathcal{L}^*[u] \;=\; \bigcup\{[\![\pi]\!]^\sharp\,\emptyset \mid \pi : u \to^* stop\}$$

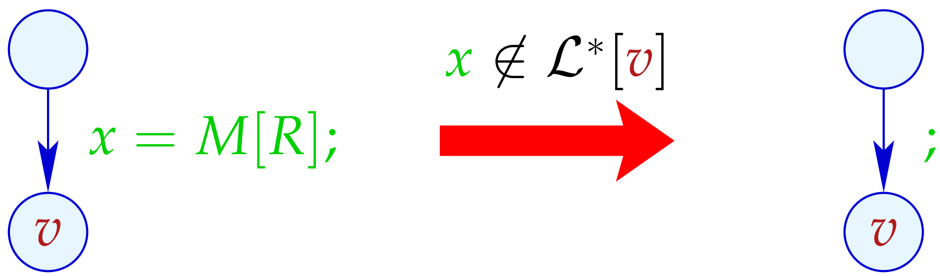
... in Worten:

- Die Pfade starten in u :-)
- x ist lebendig, wenn es nur entlang irgend eines Pfads lebendig ist :-)

  $\Longrightarrow$ Als Halbordnung für $\mathbb{L}$ benötigen wir $\sqsubseteq \, = \, \subseteq$.
- Am Programmende ist keine Variable mehr lebendig :-)

Transformation 3:





Zur Korrektheit zeigt man:

→ Korrektheit der Kanten-Effekte: Falls L die Menge der legendigen Variablen am Ende eines Pfads π sind, dann ist $[\![\pi]\!]^\sharp L$ die Menge der am Anfang lebendigen Variablen :-)

→ Korrektheit der Transformation auf einem Pfad: Wird auf den Wert einer Variable zugegriffen, ist diese stets lebendig. Der Wert toter Variablen ist darum egal :-)

→ Korrektheit der Transformation: Bei Ausführung des transformierten Programms haben bei jedem Besuch eines Programmpunkts die lebendigen Variablen den gleichen Wert :-))

Berechnung der Mengen $\mathcal{L}^*[u]$:

(1) Aufstellen des Constraint-Systems:

$$\begin{array}{lll} \mathcal{L}[stop] & \supseteq & \emptyset \\ \mathcal{L}[u] & \supseteq & [\![k]\!]^\sharp(\mathcal{L}[v]) \qquad k = (u, \_, v) \quad \text{Kante} \end{array}$$

(2) Lösen des Constraint-Systems mittels RR-Iteration.

Da $\mathbb{L}$ endlich ist, terminiert die Iteration :-)

(3) Die kleinste Lösung $\mathcal{L}$ des Constraint-Systems ist gleich $\mathcal{L}^*$ da alle $[\![k]\!]^\sharp$ distributiv sind :-))

Berechnung der Mengen $\mathcal{L}^*[u]$:

(1) Aufstellen des Constraint-Systems:

$$\begin{array}{lll} \mathcal{L}[stop] & \supseteq & \emptyset \\ \mathcal{L}[u] & \supseteq & [\![k]\!]^\sharp\,(\mathcal{L}[v]) \qquad k = (u, \_, v) \quad \text{Kante} \end{array}$$

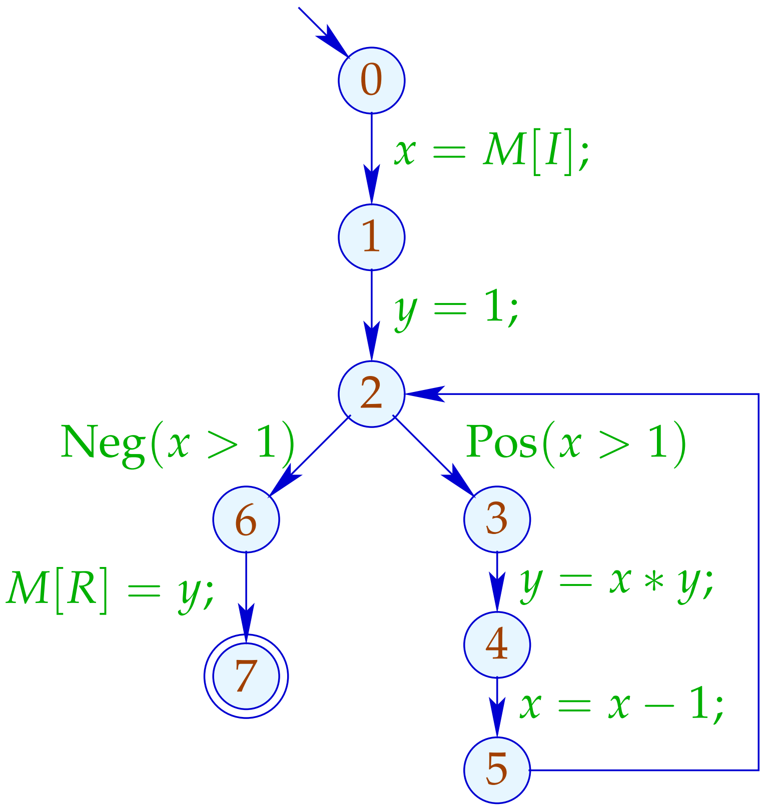
(2) Lösen des Constraint-Systems mittels RR-Iteration.

Da $\mathbb{L}$ endlich ist, terminiert die Iteration :-)

(3) Die kleinste Lösung $\mathcal{L}$ des Constraint-Systems ist gleich $\mathcal{L}^*$ da alle $[\![k]\!]^\sharp$ distributiv sind :-))

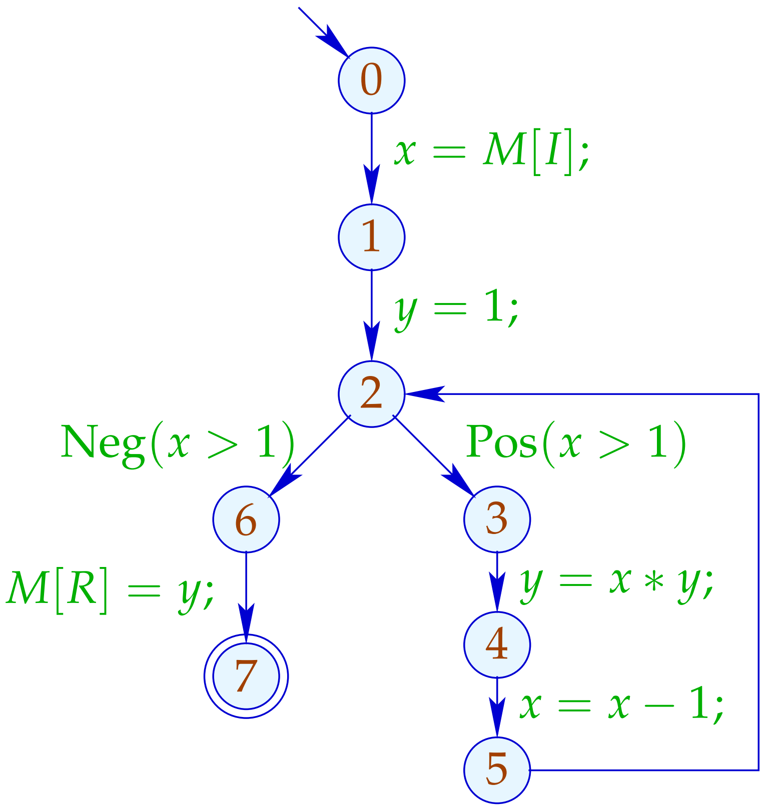
Achtung: Die Information wird rückwärts propagiert !!!

Beispiel:



$$
\begin{array}{lll}
\mathcal{L}[0] & \supseteq & (\mathcal{L}[1] \backslash \{x\}) \cup \{I\} \\
\mathcal{L}[1] & \supseteq & \mathcal{L}[2] \backslash \{y\} \\
\mathcal{L}[2] & \supseteq & (\mathcal{L}[6] \cup \{x\}) \cup (\mathcal{L}[3] \cup \{x\}) \\
\mathcal{L}[3] & \supseteq & (\mathcal{L}[4] \backslash \{y\}) \cup \{x, y\} \\
\mathcal{L}[4] & \supseteq & (\mathcal{L}[5] \backslash \{x\}) \cup \{x\} \\
\mathcal{L}[5] & \supseteq & \mathcal{L}[2] \\
\mathcal{L}[6] & \supseteq & \mathcal{L}[7] \cup \{y, R\} \\
\mathcal{L}[7] & \supseteq & \emptyset
\end{array}
$$

Beispiel:

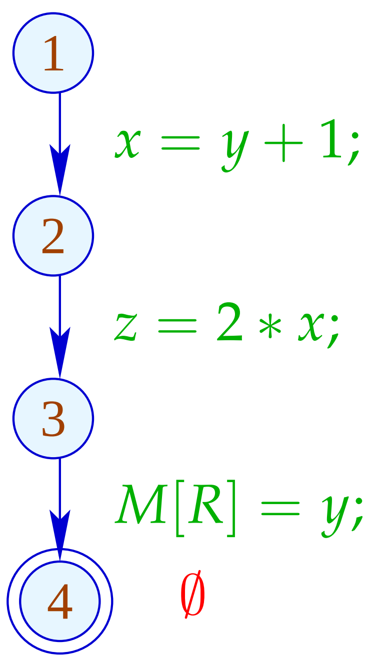


	1	2
7	$\emptyset$	
6	$\{y, R\}$	
2	$\{x, y, R\}$	dito
5	$\{x, y, R\}$	
4	$\{x, y, R\}$	
3	$\{x, y, R\}$	
1	$\{x, R\}$	
0	$\{I, R\}$	

Bei keiner Zuweisung ist die linke Variable tot :-)

Achtung:

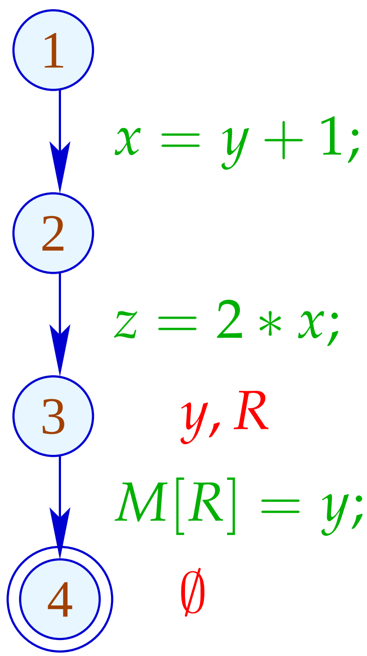
Beseitigung von Zuweisungen an tote Variablen kann weitere Variablen töten:



Bei keiner Zuweisung ist die linke Variable tot :-)

Achtung:

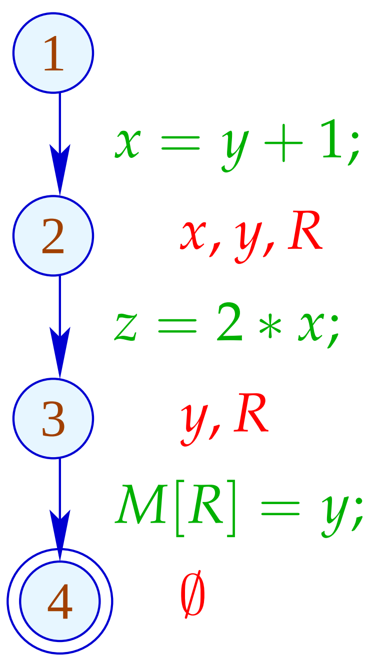
Beseitigung von Zuweisungen an tote Variablen kann weitere Variablen töten:



Bei keiner Zuweisung ist die linke Variable tot   :-)

Achtung:

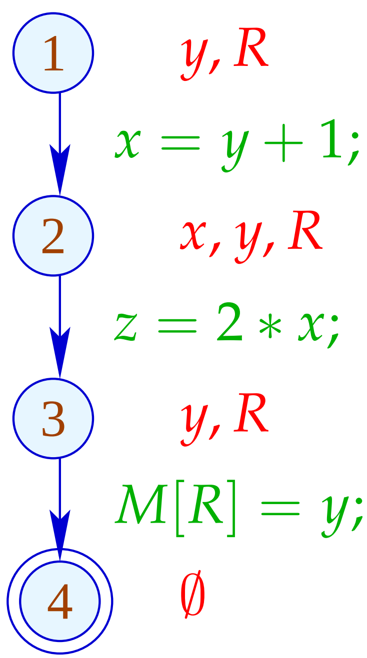
Beseitigung von Zuweisungen an tote Variablen kann weitere Variablen töten:



Bei keiner Zuweisung ist die linke Variable tot :-)

Achtung:

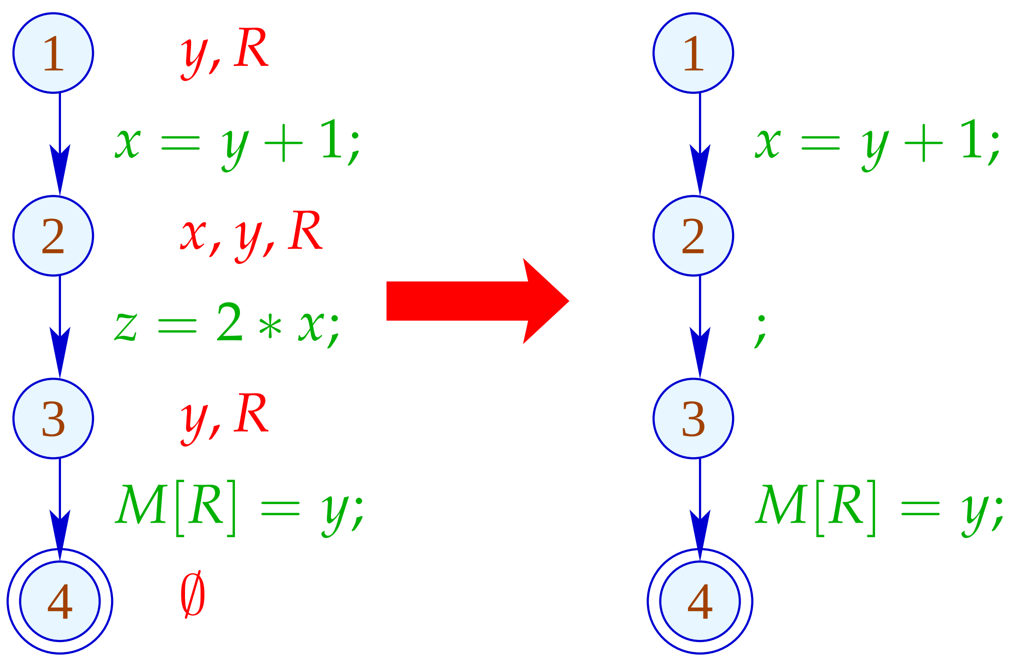
Beseitigung von Zuweisungen an tote Variablen kann weitere Variablen töten:



Bei keiner Zuweisung ist die linke Variable tot :-)

Achtung:

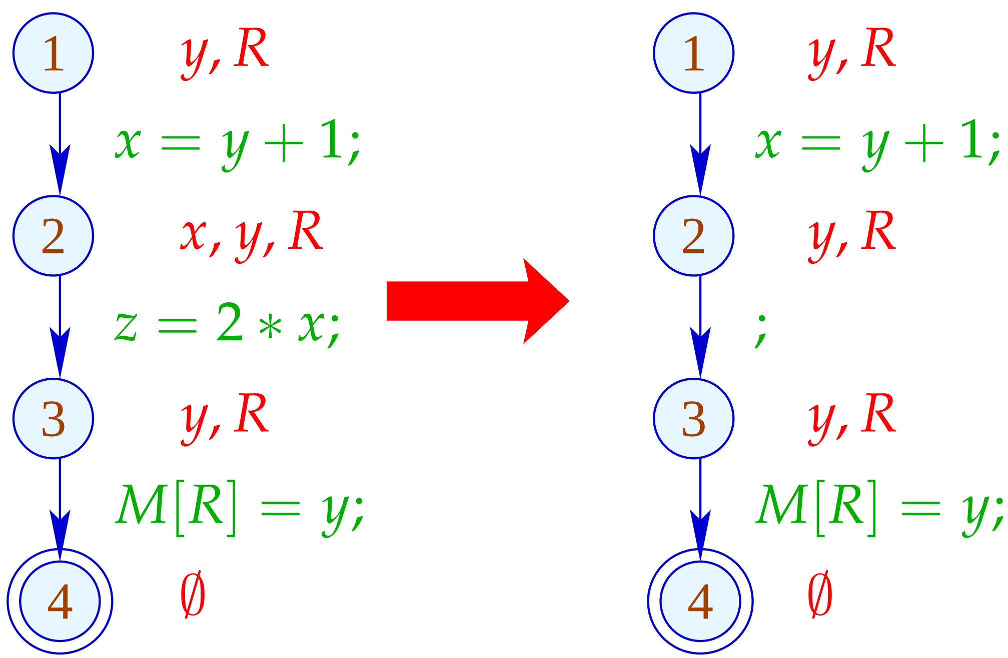
Beseitigung von Zuweisungen an tote Variablen kann weitere Variablen töten:



Bei keiner Zuweisung ist die linke Variable tot :-)

Achtung:

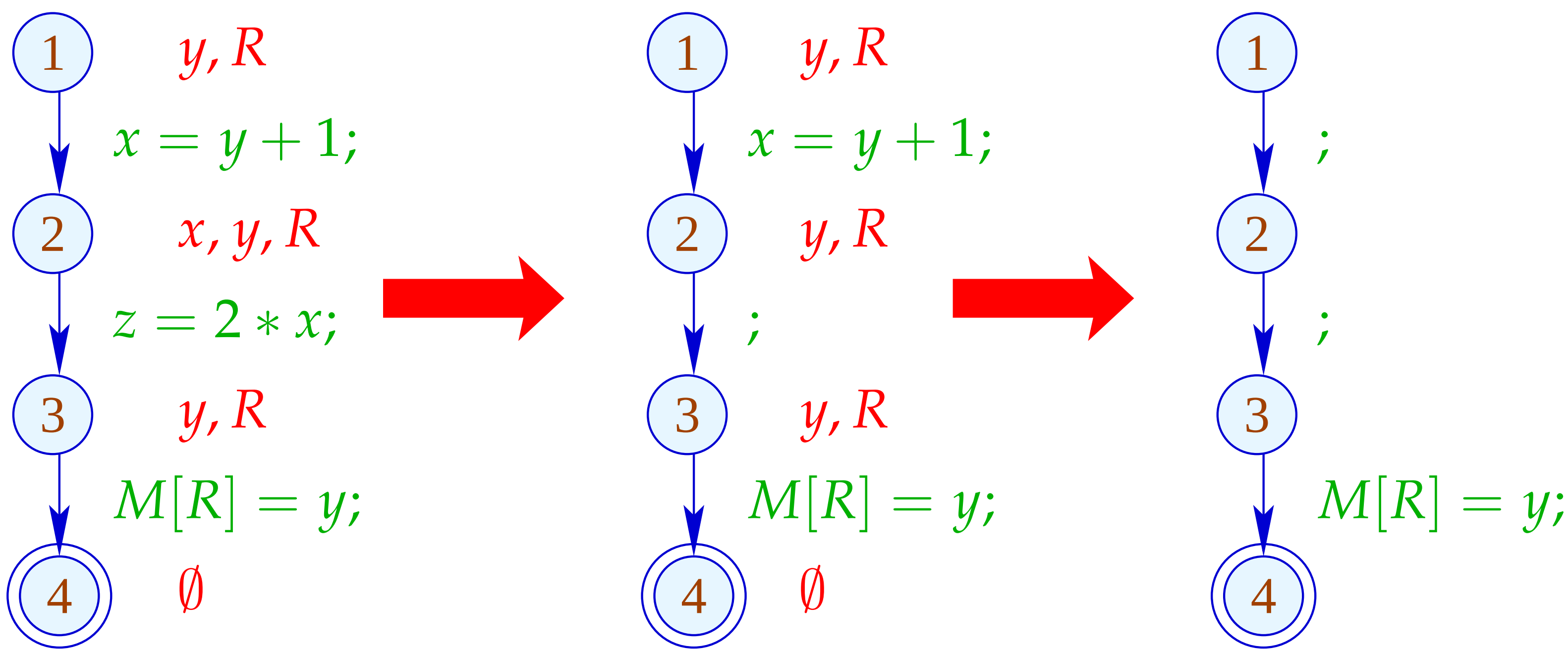
Beseitigung von Zuweisungen an tote Variablen kann weitere Variablen töten:



Bei keiner Zuweisung ist die linke Variable tot :-)

Achtung:

Beseitigung von Zuweisungen an tote Variablen kann weitere Variablen töten:

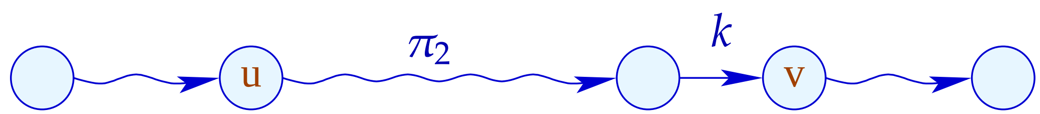


Das Programm mehrmals zu analysieren, ist hässlich :-(

Idee: Analysiere echte Lebendigkeit!

x heißt echt lebendig an u entlang eines Pfads π , falls sich π zerlegen lässt in $\pi = \pi_1\,\pi_2\,k\,\pi_3$ so dass gilt:

- π_1 erreicht u ;
- k ist eine echte Benutzung von x ;
- π_2 enthält keine Überschreibung von x.

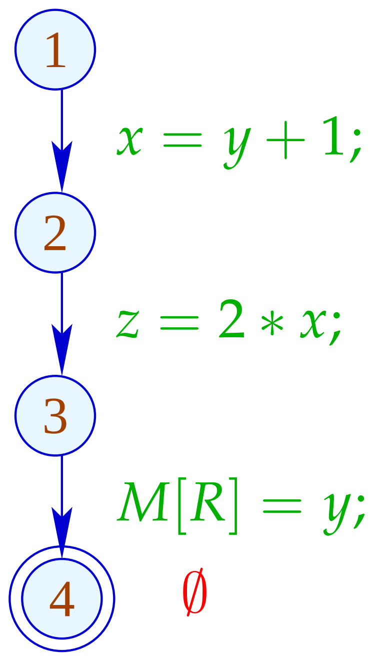


Die Menge der an einer Kante $k = (\_, lab, v)$ echt benutzten Variablen ist gegeben durch:

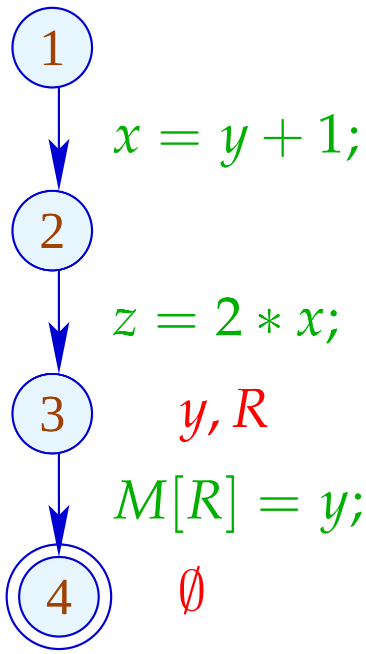
lab	echt benutzt
$;$	$\emptyset$
$\text{Pos}\,(e)$	$Vars\,(e)$
$\text{Neg}\,(e)$	$Vars\,(e)$
$x = e;$	$Vars\,(e)$ $\quad (*)$
$x = M[R];$	$\{R\}$ $\quad (*)$
$M[R_1] = R_2;$	$\{R_1, R_2\}$

$(*)$ – sofern x an v echt lebendig ist :-)

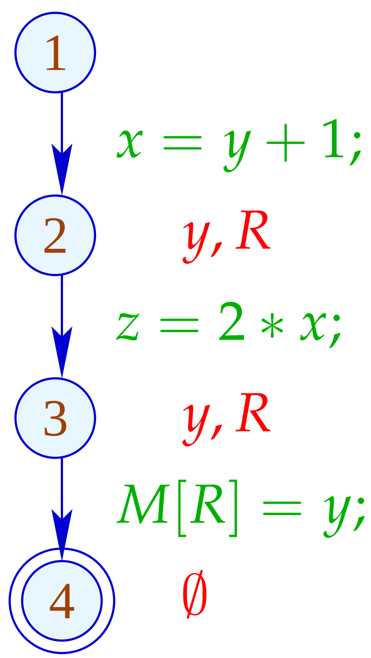
Beispiel:



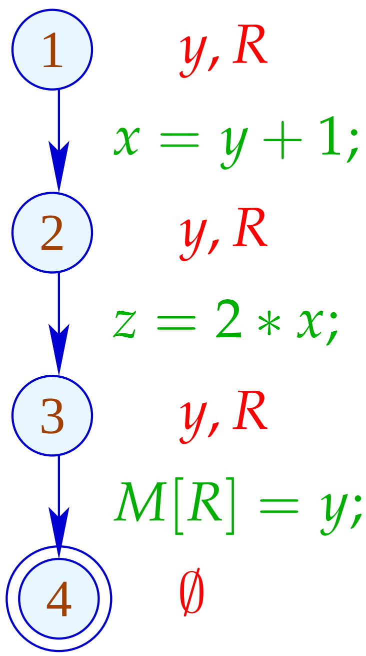
Beispiel:



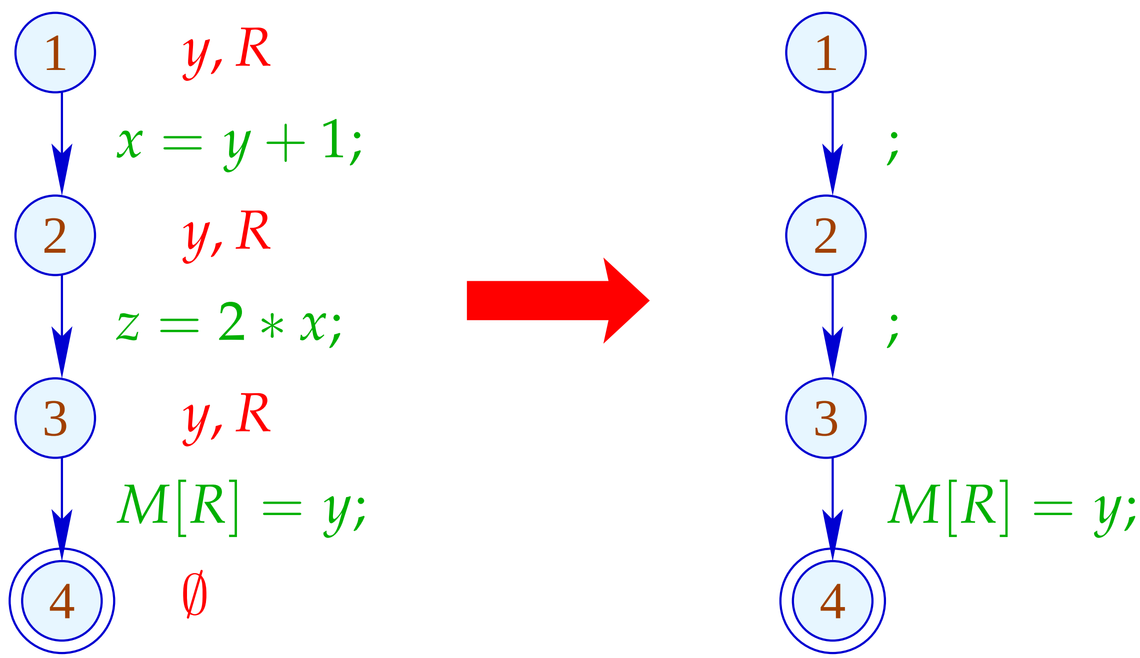
Beispiel:



Beispiel:



Beispiel:



Die Kanten-Effekte:

$$
\begin{array}{lclcl}
[\![;]\!]^\sharp\, L & = & L & & \\
[\![\text{Pos}(e)]\!]^\sharp\, L & = & [\![\text{Neg}(e)]\!]^\sharp\, L & = & L \cup \mathit{Vars}(e) \\
[\![x = e;]\!]^\sharp\, L & = & (L \backslash \{x\}) \cup & & \mathit{Vars}(e) \\
[\![R_1 = M[R_2];]\!]^\sharp\, L & = & (L \backslash \{R_1\}) \cup & & \{R_2\} \\
[\![M[R_1] = R_2;]\!]^\sharp\, L & = & L \cup \{R_1, R_2\} & &
\end{array}
$$

Die Kanten-Effekte:

$$
\begin{array}{lcl}
[\![;]\!]^\sharp\, L & = & L \\
[\![\text{Pos}(e)]\!]^\sharp\, L & = & [\![\text{Neg}(e)]\!]^\sharp\, L \quad = \quad L \cup \mathit{Vars}(e) \\
[\![x = e;]\!]^\sharp\, L & = & (L \backslash \{x\}) \cup \ (x \in L) \mathrel{?} \mathit{Vars}(e) :\ \emptyset \\
[\![R_1 = M[R_2];]\!]^\sharp\, L & = & (L \backslash \{R_1\}) \cup \ (R_1 \in L) \mathrel{?} \{R_2\} :\ \emptyset \\
[\![M[R_1] = R_2;]\!]^\sharp\, L & = & L \cup \{R_1, R_2\}
\end{array}
$$

Beachte:

- Die Kanten-Effekte für echt lebendige Variablen sind komplizierter als für lebendige Variablen :-)
- Sie sind aber immer noch distributiv !!

Beachte:

- Die Kanten-Effekte für echt lebendige Variablen sind komplizierter als für lebendige Variablen :-)
- Sie sind aber immer noch distributiv !!

  Dazu betrachten wir für $\mathbb{D} = 2^U$, $f\,y = (u \in y)\,?\,b : \emptyset$
  Wir überprüfen:

$$\begin{array}{lll} f\,(y_1 \cup y_2) & = & (u \in y_1 \cup y_2)\,?\,b : \emptyset \\ & = & (u \in y_1 \vee u \in y_2)\,?\,b : \emptyset \\ & = & (u \in y_1)\,?\,b : \emptyset \cup (u \in y_2)\,?\,b : \emptyset \\ & = & f\,y_1 \cup f\,y_2 \end{array}$$

Beachte:

- Die Kanten-Effekte für echt lebendige Variablen sind komplizierter als für lebendige Variablen :-)
- Sie sind aber immer noch distributiv !!

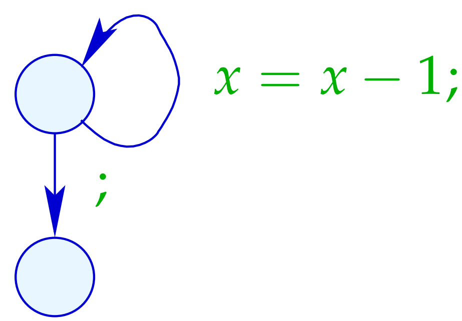
  Dazu betrachten wir für $\mathbb{D} = 2^U$, $f\,y = (u \in y)\,?\,b : \emptyset$

  Wir überprüfen:

$$
\begin{aligned}
f\,(y_1 \cup y_2) &= (u \in y_1 \cup y_2)\,?\,b : \emptyset \\
&= (u \in y_1 \vee u \in y_2)\,?\,b : \emptyset \\
&= (u \in y_1)\,?\,b : \emptyset \cup (u \in y_2)\,?\,b : \emptyset \\
&= f\,y_1 \cup f\,y_2
\end{aligned}
$$

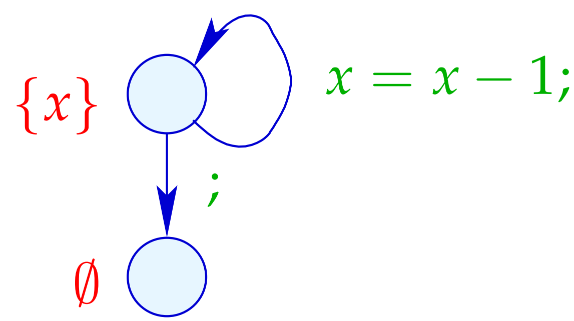
$\Longrightarrow$ Constraint-System liefert MOP :-))

- Echte Lebendigkeit findet mehr überflüssige Zuweisungen als wiederholte Lebendigkeit !!!



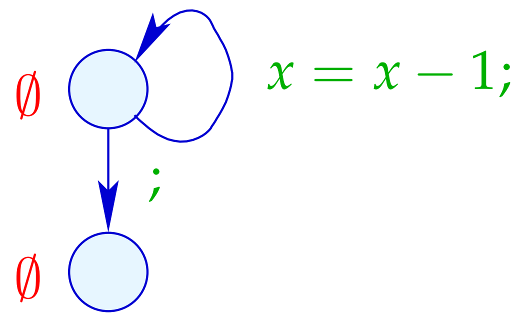
- Echte Lebendigkeit findet mehr überflüssige Zuweisungen als wiederholte Lebendigkeit !!!

Lebendigkeit:



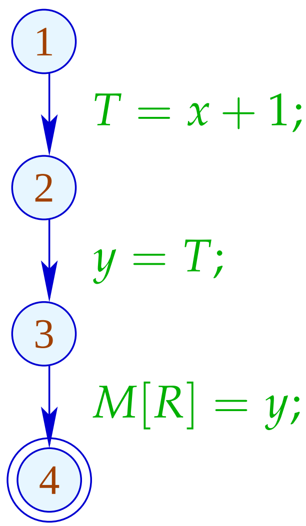
- Echte Lebendigkeit findet mehr überflüssige Zuweisungen als wiederholte Lebendigkeit !!!

Echte Lebendigkeit:



1.3 Beseitigung überflüssiger Umspeicherungen

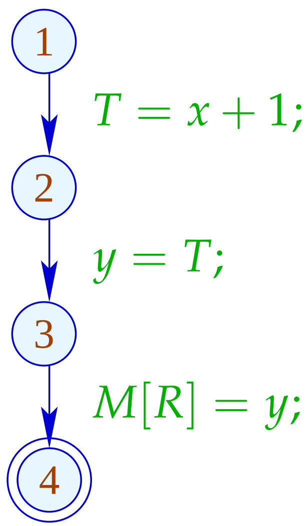
Beispiel:



Offenbar ist die Umspeicherung nutzlos :-(

1.3 Beseitigung überflüssiger Umspeicherungen

Beispiel:

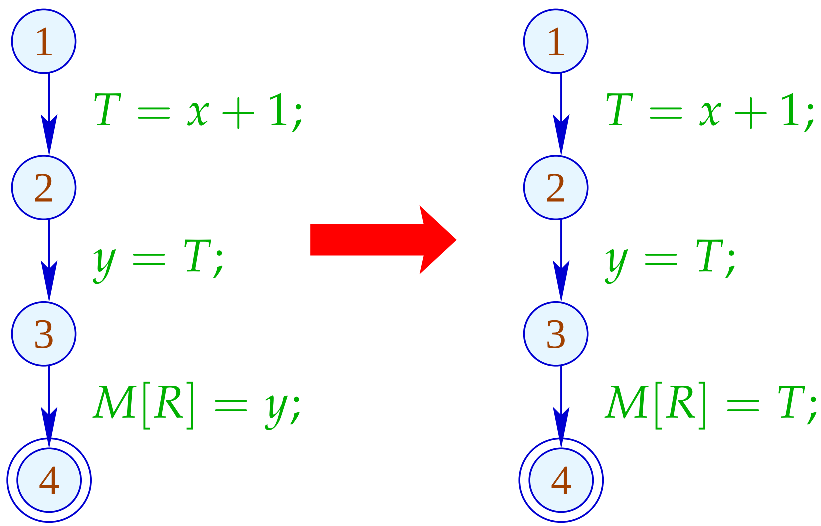


Offenbar ist die Umspeicherung nutzlos :-(

Statt y könnten wir auch T abspeichern :-)

1.3 Beseitigung überflüssiger Umspeicherungen

Beispiel:

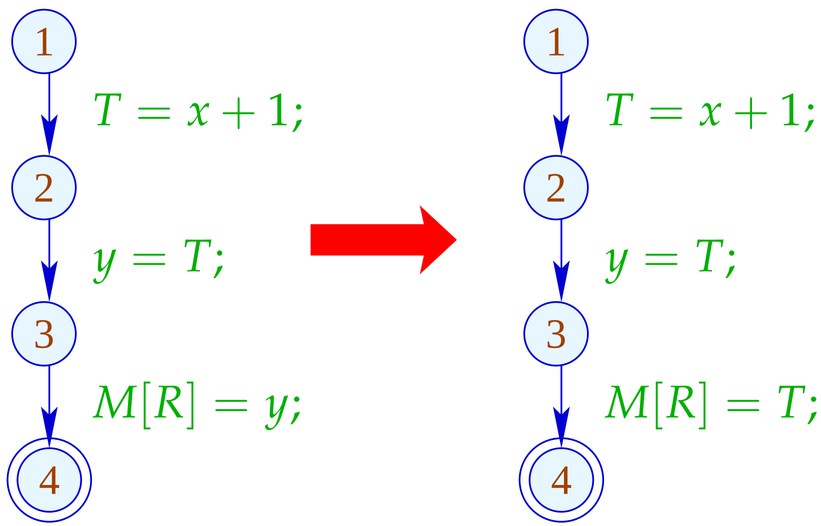


Offenbar ist die Umspeicherung nutzlos :-(

Statt y könnten wir auch T abspeichern :-)

1.3 Beseitigung überflüssiger Umspeicherungen

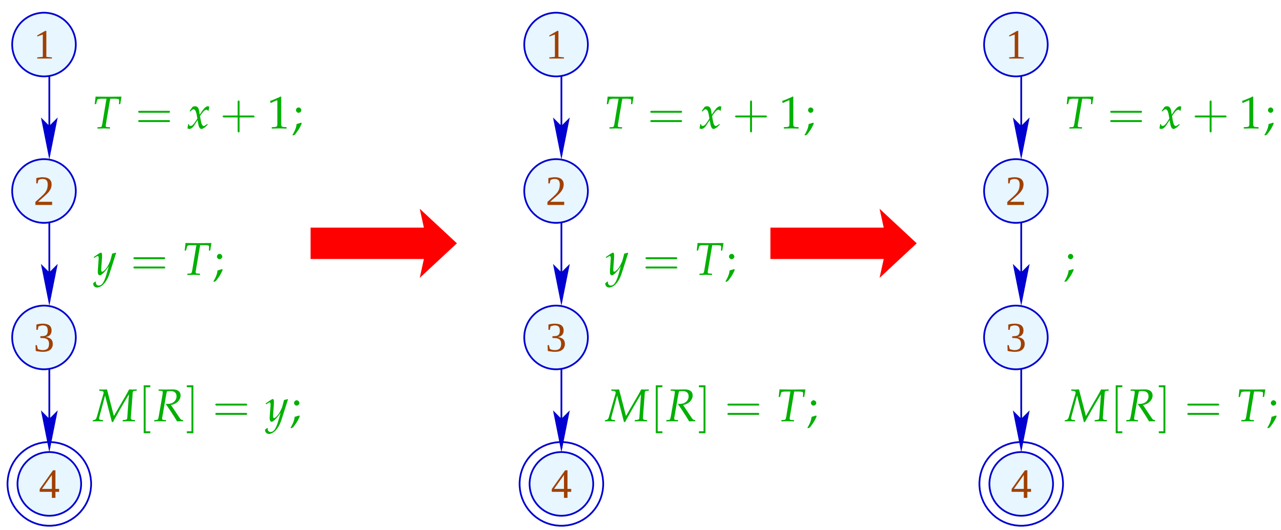
Beispiel:



Vorteil: Jetzt ist y tot :-))

1.3 Beseitigung überflüssiger Umspeicherungen

Beispiel:



Vorteil: Jetzt ist y tot :-))

Idee:

Für jeden Ausdruck merken wir uns die Variablen, die gegenwärtig seinen Wert enthalten :-)

Wir benutzen: $\mathbb{V} = \mathit{Expr} \to 2^{\mathit{Vars}}$ ...

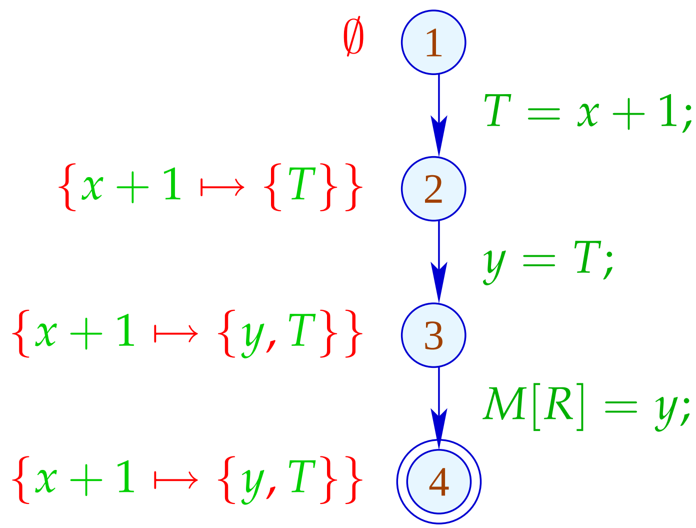
Idee:

Für jeden Ausdruck merken wir uns die Variablen, die gegenwärtig seinen Wert enthalten :-)

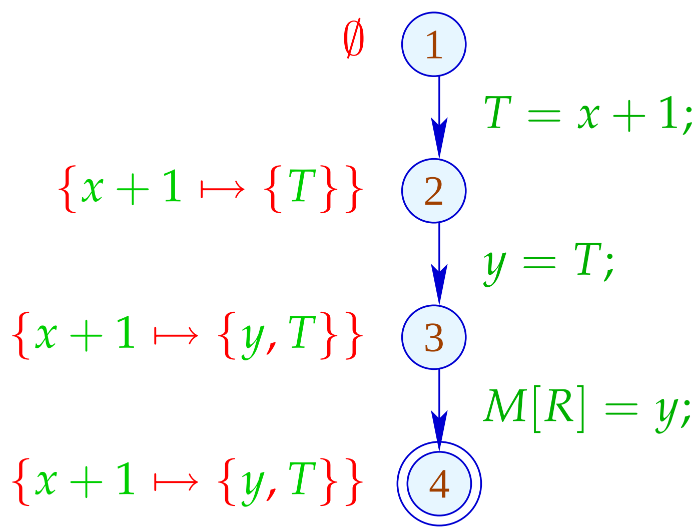
Wir benutzen: $\mathbb{V} = \mathit{Expr} \to 2^{\mathit{Vars}}$ und definieren:

$$
\begin{array}{lll}
[\![;]\!]^\sharp\, V & = & V \\
[\![\mathrm{Pos}(e)]\!]^\sharp\, V & = & [\![\mathrm{Neg}(e)]\!]^\sharp\, V \quad = \quad V \\
[\![x = e;]\!]^\sharp\, V\, e' & = & \begin{cases} \{x\} & \text{falls} \quad e' = e \\ (V\, e')\backslash\{x\} & \text{sonst} \end{cases} \\
[\![x = y;]\!]^\sharp\, V\, e & = & \begin{cases} (V\, e) \cup \{x\} & \text{falls} \quad y \in V\, e \\ (V\, e)\backslash\{x\} & \text{sonst} \end{cases} \\
[\![x = M[R];]\!]^\sharp\, V\, e & = & (V\, e)\backslash\{x\} \\
[\![M[R_1] = R_2;]\!]^\sharp\, V & = & V
\end{array}
$$

Im Beispiel:



Im Beispiel:



$\longrightarrow$ Wir propagieren die Information vorwärts :-)

An *start* haben wir $V_0\, e = \emptyset$ für alle e

$\longrightarrow$ $\sqsubseteq\ \subseteq\ \mathbb{V} \times \mathbb{V}$ definieren wir durch:

$$V_1 \sqsubseteq V_2 \quad \text{gdw.} \quad V_1\, e \quad \supseteq \quad V_2\, e \qquad \text{für alle} \quad e$$

Beobachtung:

Die neuen Kanten-Effekte sind distributiv:

Dazu zeigen wir, dass die folgenden Funktionen distributiv sind:

(1) $f_1\, V\, e = (V\, e) \backslash \{x\}$

(2) $f_2\, V = V \oplus \{e \mapsto \{x\}\}$

(3) $f_3\, V\, e = (y \in V\, e)\, ?\, (V\, e \cup \{x\}) : ((V\, e) \backslash \{x\})$

Offenbar gilt:

$$\begin{array}{lll} [\![x = e;]\!]^\sharp & = & f_2 \circ f_1 \\ [\![x = y;]\!]^\sharp & = & f_3 \\ [\![x = M[R];]\!]^\sharp & = & f_1 \end{array}$$

Distributivität ist unter Komposition abgeschlossen. Damit folgt die Behauptung :-))

(1) Für $f\, V\, e = (V\, e) \backslash \{x\}$ gilt:

$$
\begin{aligned}
f\,(V_1 \sqcup V_2)\, e &= ((V_1 \sqcup V_2)\, e) \backslash \{x\} \\
&= ((V_1\, e) \cap (V_2\, e)) \backslash \{x\} \\
&= ((V_1\, e) \backslash \{x\}) \cap ((V_2\, e) \backslash \{x\}) \\
&= (f\, V_1\, e) \cap (f\, V_2\, e) \\
&= (f\, V_1 \sqcup f\, V_2)\, e \qquad \text{:-)}
\end{aligned}
$$

(2) Für $f\,V = V \oplus \{e \mapsto a\}$ gilt:

$$
\begin{array}{lll}
f\,(V_1 \sqcup V_2)\,e' & = & ((V_1 \sqcup V_2) \oplus \{e \mapsto a\})\,e' \\
& = & (V_1 \sqcup V_2)\,e' \\
& = & (f\,V_1 \sqcup f\,V_2)\,e' \qquad \text{sofern} \quad e \neq e'
\end{array}
$$

$$
\begin{array}{lll}
f\,(V_1 \sqcup V_2)\,e & = & ((V_1 \sqcup V_2) \oplus \{e \mapsto a\})\,e \\
& = & a \\
& = & ((V_1 \oplus \{e \mapsto a\})\,e) \sqcap ((V_2 \oplus \{e \mapsto a\})\,e) \\
& = & (f\,V_1 \sqcup f\,V_2)\,e \qquad \text{:-)}
\end{array}
$$

(3) Für $f\, V\, e = (y \in V\, e)\, ?\, (V\, e \cup \{x\}) : ((V\, e) \backslash \{x\})$ gilt:

$$
\begin{aligned}
f\,(V_1 \sqcup V_2)\,e &= (((V_1 \sqcup V_2)\,e) \backslash \{x\}) \cup (y \in (V_1 \sqcup V_2)\,e)\, ?\, \{x\} : \emptyset \\
&= ((V_1\,e \cap V_2\,e) \backslash \{x\}) \cup (y \in (V_1\,e \cap V_2\,e))\, ?\, \{x\} : \emptyset \\
&= ((V_1\,e \cap V_2\,e) \backslash \{x\}) \cup \\
&\quad ((y \in V_1\,e)\, ?\, \{x\} : \emptyset) \cap ((y \in V_2\,e)\, ?\, \{x\} : \emptyset) \\
&= (((V_1\,e) \backslash \{x\}) \cup (y \in V_1\,e)\, ?\, \{x\} : \emptyset) \cap \\
&\quad (((V_2\,e) \backslash \{x\}) \cup (y \in V_2\,e)\, ?\, \{x\} : \emptyset) \\
&= (f\,V_1 \sqcup f\,V_2)\,e \qquad\qquad \text{:-)}
\end{aligned}
$$

Wir schließen:

$\rightarrow$ Lösen des Constraint-Systems liefert die MOP-Lösung :-)

$\rightarrow$ Sei $\mathcal{V}$ diese Lösung.

Gilt $x \in \mathcal{V}[u]\, e$, enthält x an u den Wert von e — welchen wir in T_e abgespeichert haben

$\Longrightarrow$

der Zugriff auf x kann durch Zugriff auf T_e ersetzt werden :-)

Für $V \in \mathbb{V}$ sei V^- die Variablen-Substitution mit:

$$V^- \, x = \begin{cases} T_e & \text{falls } x \in V\, e \\ x & \text{sonst} \end{cases}$$

falls $V\, e \cap V\, e' = \emptyset$ für $e \neq e'$. Andernfalls: $V^- \, x = x$:-)

Transformation 4:

$$\begin{tikzcd} u \arrow[d, "\mathrm{Pos}\,(e)"] \\ \circ \end{tikzcd} \quad \xrightarrow{\sigma = \mathcal{V}[u]^{-}} \quad \begin{tikzcd} u \arrow[d, "\mathrm{Pos}\,(\sigma(e))"] \\ \circ \end{tikzcd}$$

... analog für Kanten mit $\mathrm{Neg}\,(e)$

$$\begin{tikzcd} u \arrow[d, "x = e;"] \\ \circ \end{tikzcd} \quad \xrightarrow{\sigma = \mathcal{V}[u]^{-}} \quad \begin{tikzcd} u \arrow[d, "x = \sigma(e);"] \\ \circ \end{tikzcd}$$

Transformation 4 (Forts.):

$$\begin{tikzcd} u \arrow[d, "x = M[R];"] \\ {} \end{tikzcd} \quad \xRightarrow{\sigma = \mathcal{V}[u]^-} \quad \begin{tikzcd} u \arrow[d, "x = M[\sigma(R)];"] \\ {} \end{tikzcd}$$

$$\begin{tikzcd} u \arrow[d, "M[R_1] = R_2;"] \\ {} \end{tikzcd} \quad \xRightarrow{\sigma = \mathcal{V}[u]^-} \quad \begin{tikzcd} u \arrow[d, "M[\sigma(R_1)] = \sigma(R_2);"] \\ {} \end{tikzcd}$$

Vorgehen insgesamt:

(1) Verfügbarkeit von Ausdrücken: T1 + T2

+ verringert arithmetische Operationen
– fügt überflüssige Umspeicherungen ein

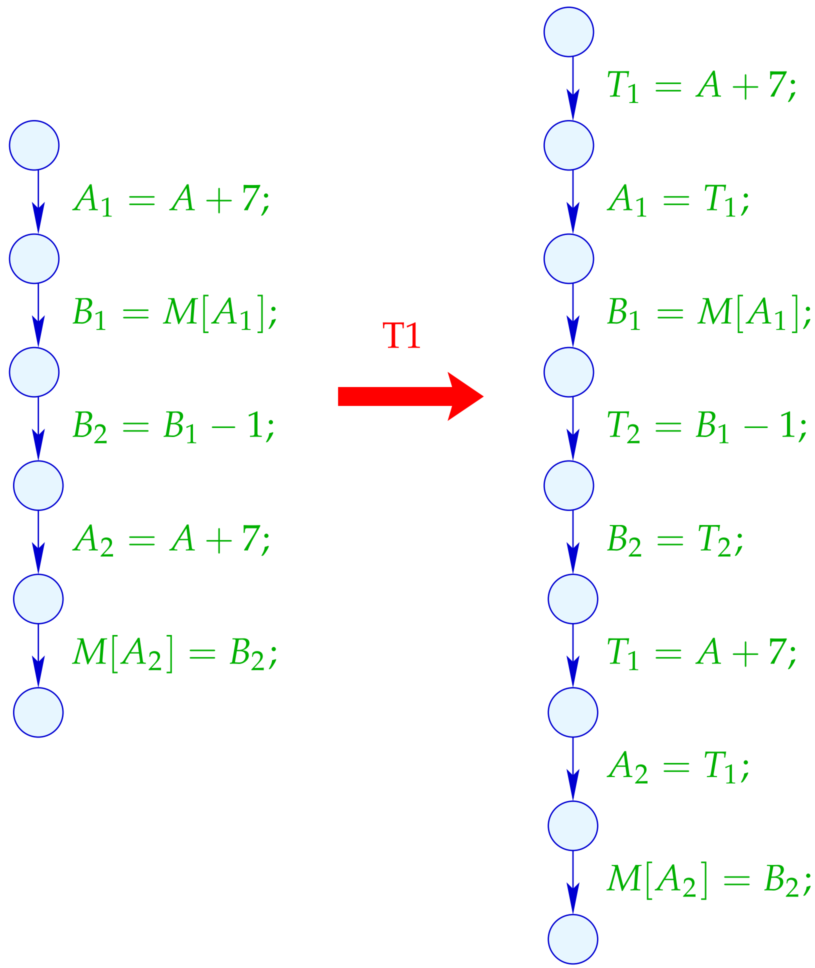
(2) Werte von Variablen: T4

+ erzeugt tote Variablen

(3) (wahre) Lebendigkeit von Variablen: T3

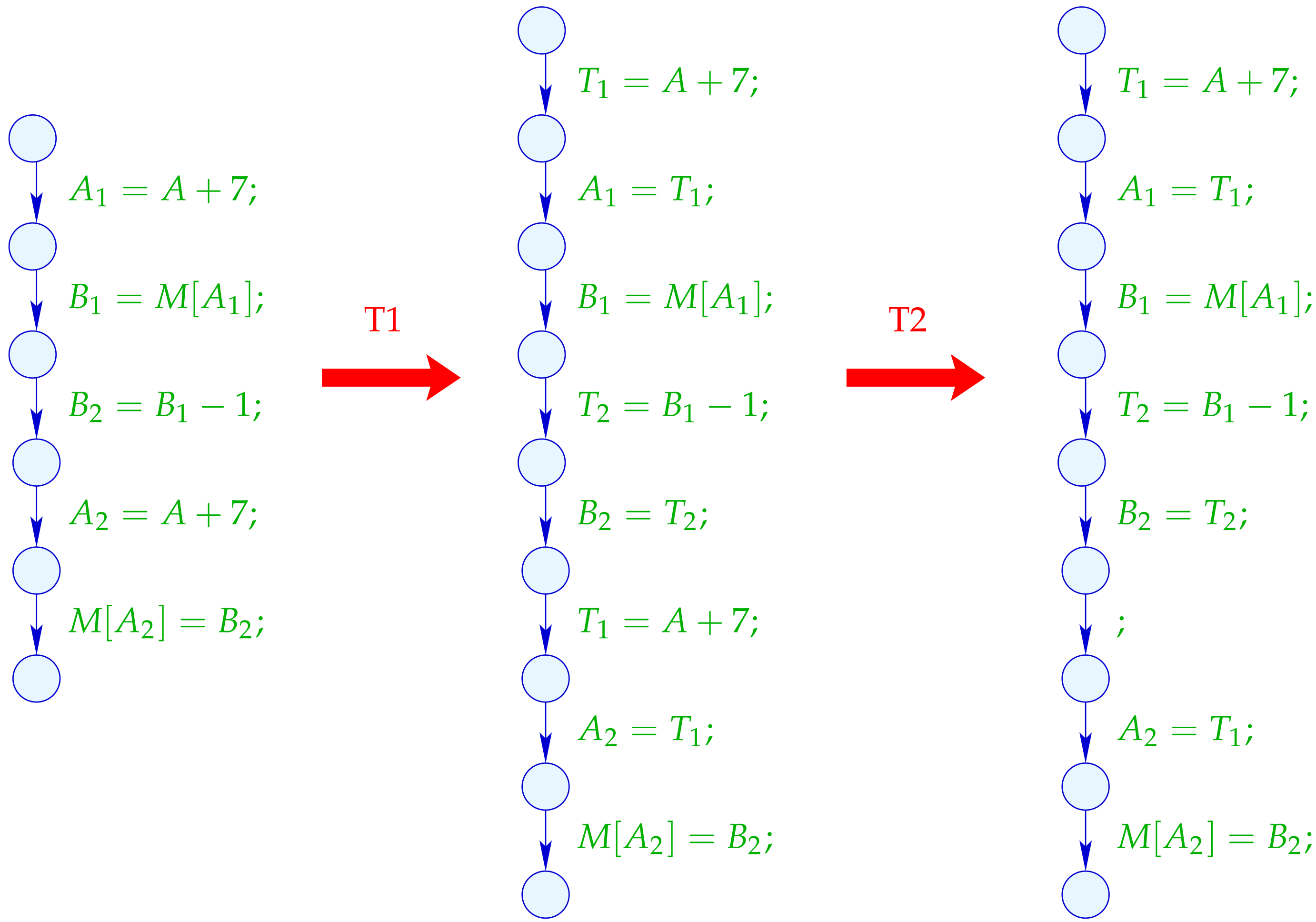
+ beseitigt Zuweisungen an tote Variablen

Beispiel: `a[7]--;`



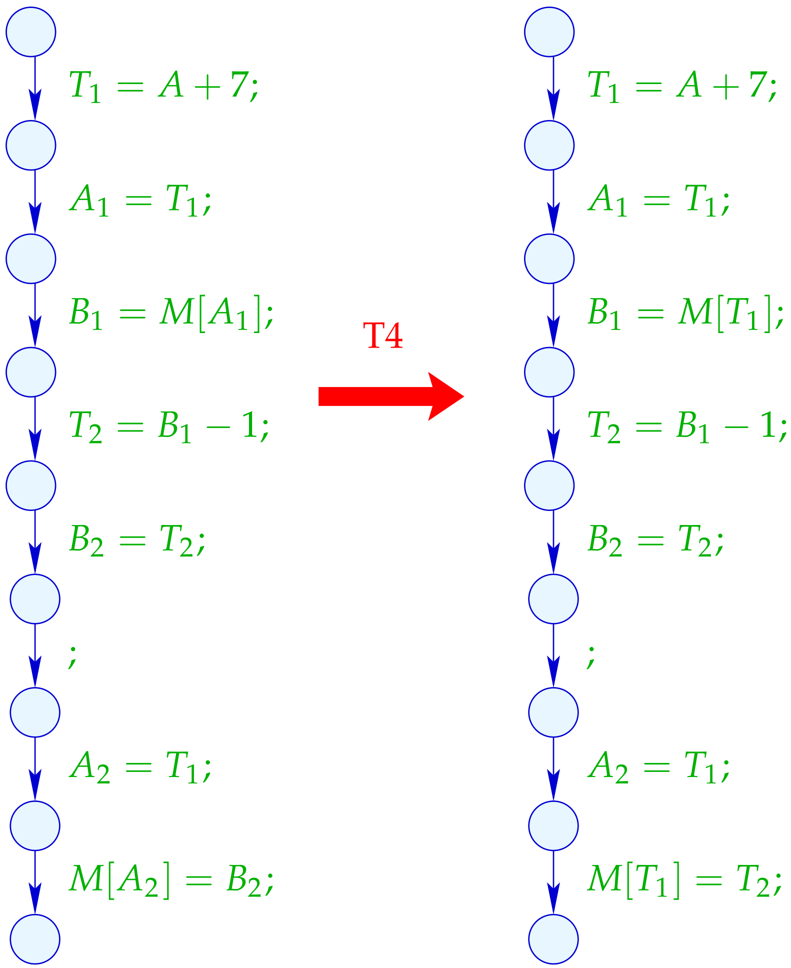
$A_1 = A + 7;$
$B_1 = M[A_1];$
$B_2 = B_1 - 1;$
$A_2 = A + 7;$
$M[A_2] = B_2;$
T1
$T_1 = A + 7;$
$A_1 = T_1;$
$B_1 = M[A_1];$
$T_2 = B_1 - 1;$
$B_2 = T_2;$
$T_1 = A + 7;$
$A_2 = T_1;$
$M[A_2] = B_2;$

Beispiel: `a[7]--;`

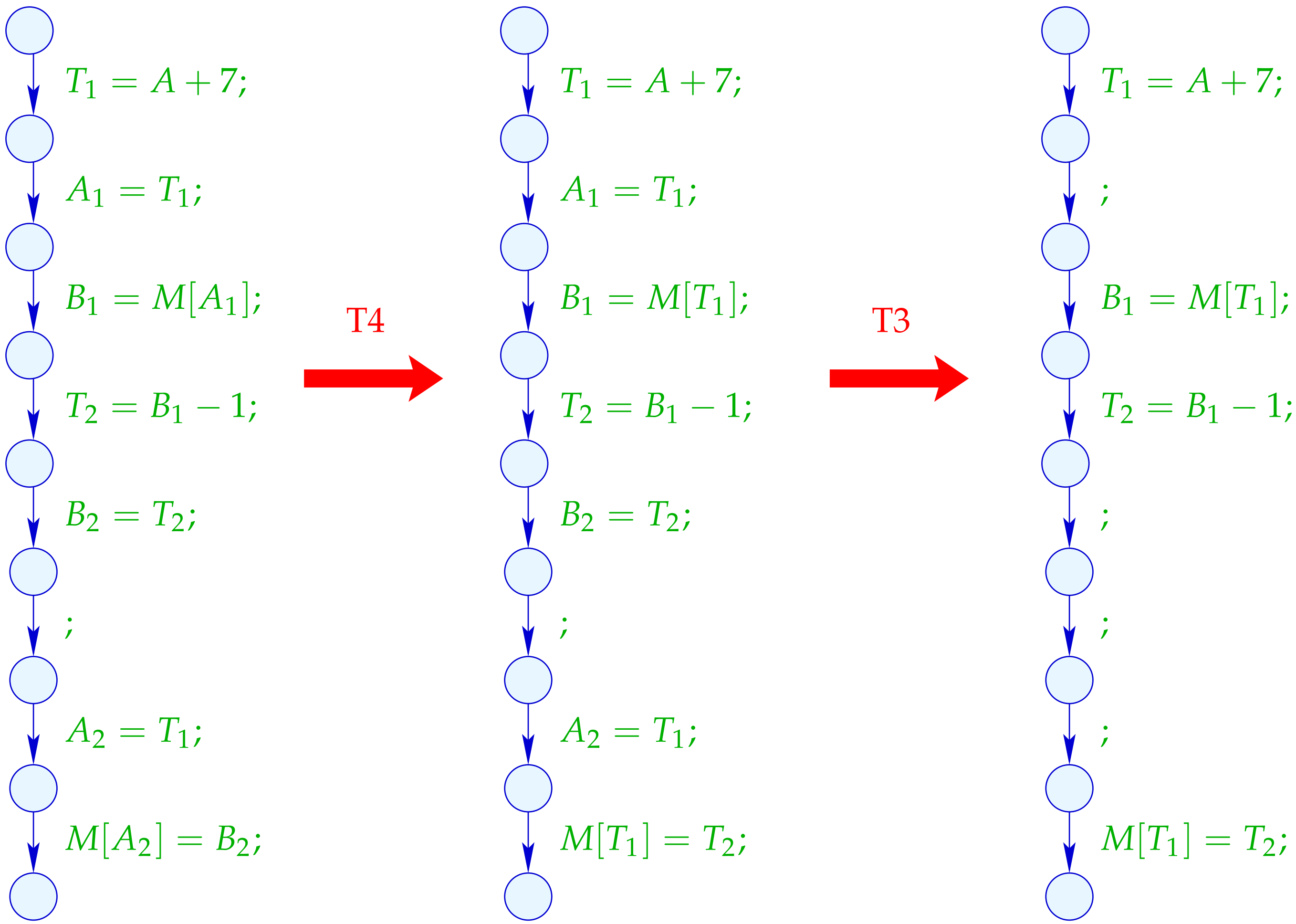


$A_1 = A + 7;$
$B_1 = M[A_1];$
$B_2 = B_1 - 1;$
$A_2 = A + 7;$
$M[A_2] = B_2;$
T1
$T_1 = A + 7;$
$A_1 = T_1;$
$B_1 = M[A_1];$
$T_2 = B_1 - 1;$
$B_2 = T_2;$
$T_1 = A + 7;$
$A_2 = T_1;$
$M[A_2] = B_2;$
T2
$T_1 = A + 7;$
$A_1 = T_1;$
$B_1 = M[A_1];$
$T_2 = B_1 - 1;$
$B_2 = T_2;$
;
$A_2 = T_1;$
$M[A_2] = B_2;$

Beispiel (Forts.): `a[7]--;`



Beispiel (Forts.): `a[7]--;`



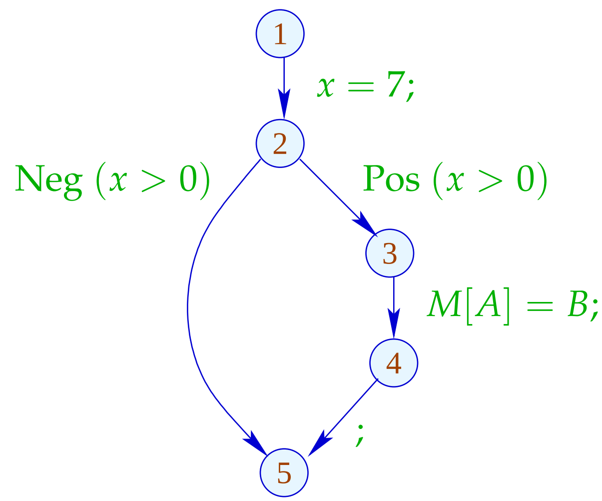
1.4 Konstanten-Propagation

Idee:

Führe möglichst große Teile des Codes bereits zur Compilezeit aus!

Beispiel:

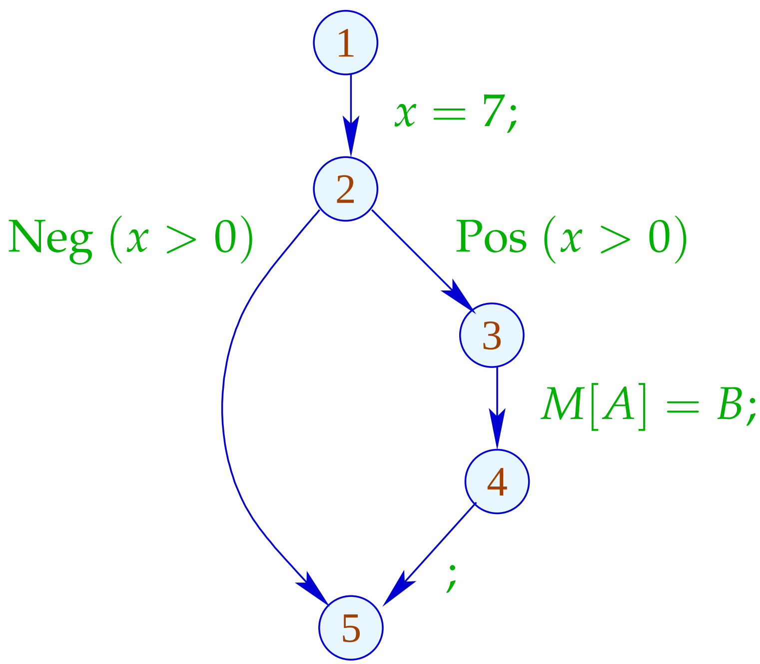
$$
\begin{array}{l}
x = 7; \\
\text{if } (x > 0) \\
\qquad M[A] = B;
\end{array}
$$



Offenbar hat x stets den Wert 7 :-)

Deshalb wird stets der Speicherzugriff durchgeführt :-))

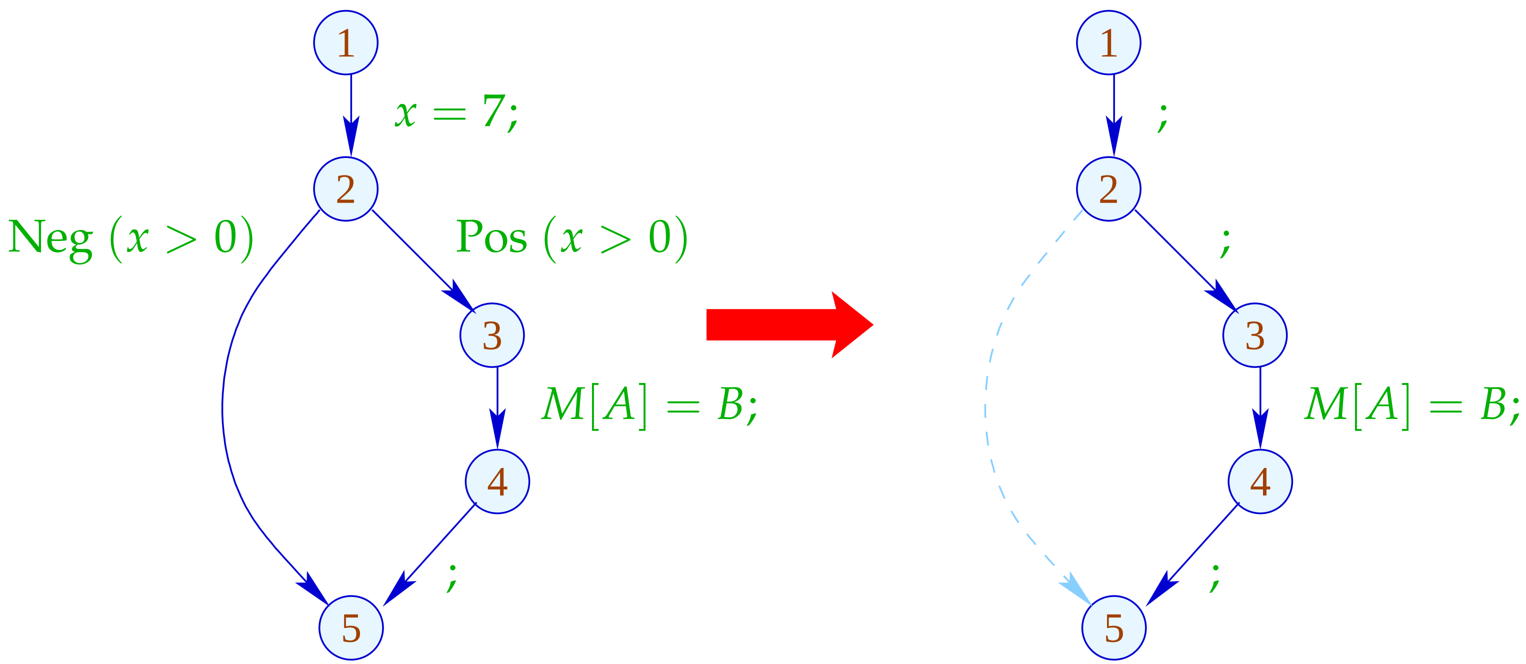
Ziel:



Offenbar hat x stets den Wert 7 :-)

Deshalb wird stets der Speicherzugriff durchgeführt :-))

Ziel:



Verallgemeinerung: Partielle Auswertung

Neil D. Jones, DIKU, Kopenhagen

Idee:

Entwerfe eine Analyse, die für jedes u

- die Werte ermittelt, die Variablen sicher haben;
- mitteilt, ob u überhaupt erreichbar ist :-)

Idee:

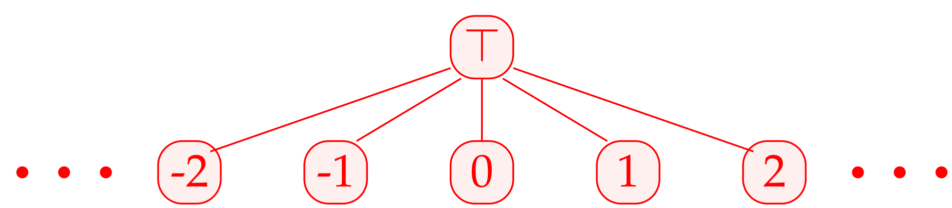
Entwerfe eine Analyse, die für jedes u

- die Werte ermittelt, die Variablen sicher haben;
- mitteilt, ob u überhaupt erreichbar ist :-)

Den vollständigen Verband konstruieren wir in zwei Schritten.

(1) Die möglichen Werte für Variablen:

$$\mathbb{Z}^\top = \mathbb{Z} \cup \{\top\} \qquad \text{mit} \quad x \sqsubseteq y \quad \text{gdw. } y = \top \text{ oder } x = y$$



Achtung: $\mathbb{Z}^\top$ ist selbst kein vollständiger Verband :-(

(2) $\mathbb{D} = (\mathit{Vars} \to \mathbb{Z}^\top)_\perp = (\mathit{Vars} \to \mathbb{Z}^\top) \cup \{\perp\}$

// $\perp$ heißt: "nicht erreichbar" :-))

mit $D_1 \sqsubseteq D_2$ gdw. $\perp = D_1$ oder

$D_1\, x \sqsubseteq D_2\, x \quad (x \in \mathit{Vars})$

Bemerkung: $\mathbb{D}$ ist ein vollständiger Verband :-)

Achtung: $\mathbb{Z}^\top$ ist selbst kein vollständiger Verband :-(

(2) $\mathbb{D} = (\mathit{Vars} \to \mathbb{Z}^\top)_\perp = (\mathit{Vars} \to \mathbb{Z}^\top) \cup \{\perp\}$

// $\perp$ heißt: “nicht erreichbar” :-))

mit $D_1 \sqsubseteq D_2$ gdw. $\perp = D_1$ oder

$D_1\,x \sqsubseteq D_2\,x \quad (x \in \mathit{Vars})$

Bemerkung: $\mathbb{D}$ ist ein vollständiger Verband :-)

Betrachte dazu $X \subseteq \mathbb{D}$. O.E. $\perp \not\in X$.

Dann $X \subseteq \mathit{Vars} \to \mathbb{Z}^\top$.

Ist $X = \emptyset$, dann $\bigsqcup X = \perp \in \mathbb{D}$:-)

Ist $X \neq \emptyset$, dann ist $\bigsqcup X = D$ mit

$$
\begin{aligned}
D\,x &= \bigsqcup\{f\,x \mid f \in X\} \\
&= \begin{cases} z & \text{falls} \quad f\,x = z \quad (f \in X) \\ \top & \text{sonst} \end{cases}
\end{aligned}
$$

:-))

Ist $X \neq \emptyset$, dann ist $\bigsqcup X = D$ mit

$$D\,x \;=\; \bigsqcup\{f\,x \mid f \in X\}$$
$$= \begin{cases} z & \text{falls} \quad f\,x = z \quad (f \in X) \\ \top & \text{sonst} \end{cases}$$

:-))

Zu jeder Kante $k = (\_, lab, \_)$ konstruieren wir eine Effekt-Funktion $[\![k]\!]^\sharp = [\![lab]\!]^\sharp : \mathbb{D} \to \mathbb{D}$, die die konkrete Berechnung simuliert.

Offenbar ist $[\![lab]\!]^\sharp\, \bot = \bot$ für alle lab :-)

Sei darum nun $\bot \neq D \in Vars \to \mathbb{Z}^\top$.

Idee:

- Wir benutzen D, um die Werte von Ausdrücken zu ermitteln.

Idee:

- Wir benutzen D , um die Werte von Ausdrücken zu ermitteln.
- Für manche Teilausdrücke erhalten wir $\top$:-)

Idee:

- Wir benutzen D , um die Werte von Ausdrücken zu ermitteln.
- Für manche Teilausdrücke erhalten wir $\top$:-)

  $\Longrightarrow$

  Wir müssen die konkreten Operatoren $\Box$ durch abstrakte Operatoren $\Box^\sharp$ ersetzen, die mit $\top$ umgehen können:

$$a \,\Box^\sharp\, b = \begin{cases} \top & \text{falls} \quad a = \top \text{ oder } b = \top \\ a \,\Box\, b & \text{sonst} \end{cases}$$

Idee:

- Wir benutzen D , um die Werte von Ausdrücken zu ermitteln.
- Für manche Teilausdrücke erhalten wir $\top$:-)

  $\Longrightarrow$

  Wir müssen die konkreten Operatoren $\Box$ durch abstrakte Operatoren $\Box^\sharp$ ersetzen, die mit $\top$ umgehen können:

$$a \,\Box^\sharp\, b = \begin{cases} \top & \text{falls} \quad a = \top \text{ oder } b = \top \\ a \,\Box\, b & \text{sonst} \end{cases}$$

- Mit den abstrakten Operatoren können wir eine abstrakte Ausdrucks-Auswertung definieren:

$$[\![e]\!]^\sharp \;:\; (\mathit{Vars} \to \mathbb{Z}^\top) \to \mathbb{Z}^\top$$

Abstrakte Ausdrucksauswertung ist wie konkrete Ausdrucksauswertung, aber mit abstrakten Werten und Operatoren. Hier:

$$\begin{array}{lll}
[\![c]\!]^\sharp\, D & = & c \\
[\![e_1 \,\square\, e_2]\!]^\sharp\, D & = & [\![e_1]\!]^\sharp\, D \,\square^\sharp\, [\![e_2]\!]^\sharp\, D
\end{array}$$

... analog für unäre Operatoren :-)

Abstrakte Ausdrucksauswertung ist wie konkrete Ausdrucksauswertung, aber mit abstrakten Werten und Operatoren. Hier:

$$\begin{array}{lll}
[\![c]\!]^\sharp\, D & = & c \\
[\![e_1 \,\Box\, e_2]\!]^\sharp\, D & = & [\![e_1]\!]^\sharp\, D \,\Box^\sharp\, [\![e_2]\!]^\sharp\, D
\end{array}$$

... analog für unäre Operatoren :-)

Beispiel: $D = \{x \mapsto 2, y \mapsto \top\}$

$$\begin{array}{lll}
[\![x + 7]\!]^\sharp\, D & = & [\![x]\!]^\sharp\, D +^\sharp [\![7]\!]^\sharp\, D \\
 & = & 2 +^\sharp 7 \\
 & = & 9 \\
[\![x - y]\!]^\sharp\, D & = & 2 -^\sharp \top \\
 & = & \top
\end{array}$$

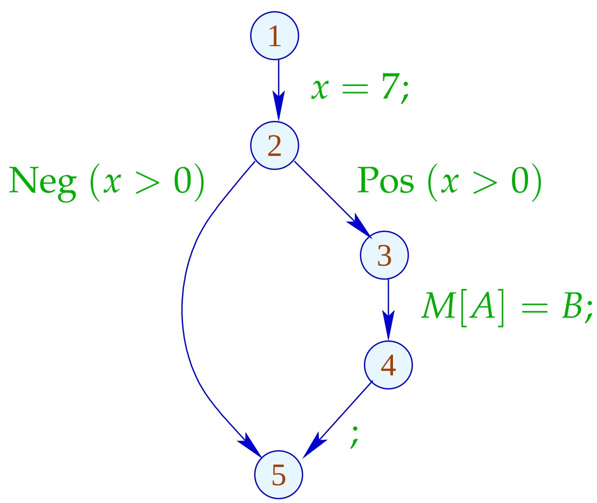
Damit erhalten wir für die Kanten-Effekte $[\![lab]\!]^\sharp$:

$$
\begin{array}{lll}
[\![;]\!]^\sharp\, D & = & D \\
[\![\text{Pos}\,(e)]\!]^\sharp\, D & = & \left\{\begin{array}{lll} \bot & \text{falls} & 0 = [\![e]\!]^\sharp\, D \\ D & \text{sonst} & \end{array}\right. \\
[\![\text{Neg}\,(e)]\!]^\sharp\, D & = & \left\{\begin{array}{lll} D & \text{falls} & 0 \sqsubseteq [\![e]\!]^\sharp\, D \\ \bot & \text{sonst} & \end{array}\right. \\
[\![x = e;]\!]^\sharp\, D & = & D \oplus \{x \mapsto [\![e]\!]^\sharp\, D\} \\
[\![x = M[R];]\!]^\sharp\, D & = & D \oplus \{x \mapsto \top\} \\
[\![M[R_1] = R_2;]\!]^\sharp\, D & = & D
\end{array}
$$

... sofern $\quad D \neq \bot \quad$:-)

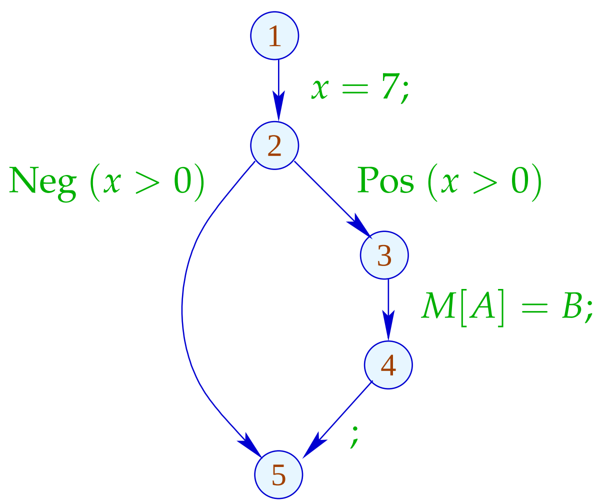
An *start* gilt $D_\perp = \{x \mapsto \top \mid x \in \mathit{Vars}\}$.

Beispiel:



An $\quad start \quad$ gilt $\quad D_\perp = \{x \mapsto \top \mid x \in Vars\}$.

Beispiel:



1	$\{x \mapsto \top\}$
2	$\{x \mapsto 7\}$
3	$\{x \mapsto 7\}$
4	$\{x \mapsto 7\}$
5	$\perp \sqcup \{x \mapsto 7\} = \{x \mapsto 7\}$

Die abstrakten Kanten-Effekte $[\![k]\!]^\sharp$ setzen wir wieder zu den Effekten von Pfaden $\pi = k_1 \ldots k_r$ zusammen durch:

$$[\![\pi]\!]^\sharp = [\![k_r]\!]^\sharp \circ \ldots \circ [\![k_1]\!]^\sharp \quad : \mathbb{D} \to \mathbb{D}$$

Idee zur Korrektheit: Abstrakte Interpretation

Cousot, Cousot 1977

Patrick Cousot, ENS, Paris

Die abstrakten Kanten-Effekte $[\![k]\!]^\sharp$ setzen wir wieder zu den Effekten von Pfaden $\pi = k_1 \ldots k_r$ zusammen durch:

$$[\![\pi]\!]^\sharp = [\![k_r]\!]^\sharp \circ \ldots \circ [\![k_1]\!]^\sharp \quad : \mathbb{D} \to \mathbb{D}$$

Idee zur Korrektheit: Abstrakte Interpretation

Cousot, Cousot 1977

Aufstellen einer Beschreibungsrelation Δ zwischen konkreten Werten und deren Beschreibungen mit:

$$x \,\Delta\, a_1 \quad \wedge \quad a_1 \sqsubseteq a_2 \quad \Longrightarrow \quad x \,\Delta\, a_2$$

Konkretisierung: $\gamma\, a = \{x \mid x \,\Delta\, a\}$

// liefert Menge der beschriebenen Werte :-)

(1) Werte: $\Delta \subseteq \mathbb{Z} \times \mathbb{Z}^\top$

$$z \,\Delta\, a \quad \text{gdw.} \quad z = a \vee a = \top$$

Konkretisierung:

$$\gamma\, a = \begin{cases} \{a\} & \text{falls} \quad a \sqsubset \top \\ \mathbb{Z} & \text{falls} \quad a = \top \end{cases}$$

(1) Werte: $\Delta \subseteq \mathbb{Z} \times \mathbb{Z}^\top$

$$z \,\Delta\, a \quad \text{gdw.} \quad z = a \vee a = \top$$

Konkretisierung:

$$\gamma\, a = \begin{cases} \{a\} & \text{falls} \quad a \sqsubset \top \\ \mathbb{Z} & \text{falls} \quad a = \top \end{cases}$$

(2) Variablenbelegungen: $\Delta \subseteq (\mathit{Vars} \to \mathbb{Z}) \times (\mathit{Vars} \to \mathbb{Z}^\top)_\perp$

$$\rho \,\Delta\, D \quad \text{gdw.} \quad D \neq \perp \wedge \rho\, x \sqsubseteq D\, x \quad (x \in \mathit{Vars})$$

Konkretisierung:

$$\gamma\, D = \begin{cases} \emptyset & \text{falls} \quad D = \perp \\ \{\rho \mid \forall\, x : \ (\rho\, x) \,\Delta\, (D\, x)\} & \text{sonst} \end{cases}$$

Beispiel: $\{x \mapsto 1, y \mapsto -7\} \;\Delta\; \{x \mapsto \top, y \mapsto -7\}$

(3) Zustände:

$$\Delta \;\subseteq\; ((\mathit{Vars} \to \mathbb{Z}) \times (\mathbb{N} \to \mathbb{Z})) \times (\mathit{Vars} \to \mathbb{Z}^{\top})_{\perp}$$

$$(\rho, \mu) \;\Delta\; D \qquad \text{gdw.} \qquad \rho \;\Delta\; D$$

Konkretisieruing:

$$\gamma\, D = \begin{cases} \emptyset & \text{falls} \quad D = \perp \\ \{(\rho, \mu) \mid \forall\, x : \; (\rho\, x) \;\Delta\; (D\, x)\} & \text{sonst} \end{cases}$$

Wir zeigen:

$(*)$ Gilt $s \, \Delta \, D$ und ist $[\![\pi]\!] \, s$ definiert, dann gilt auch:

$$([\![\pi]\!] \, s) \;\; \Delta \;\; ([\![\pi]\!]^\sharp \, D)$$

$$\begin{tikzcd}
\boxed{s} \arrow[rrr, "{[\![\pi]\!]}"] \arrow[d, dash, "\Delta"'] & & & \boxed{s_1} \arrow[d, dash, dotted, "\Delta"] \\
\boxed{D} \arrow[rrr, "{[\![\pi]\!]^\sharp}"'] & & & \boxed{D_1}
\end{tikzcd}$$

Die abstrakte Semantik simuliert die konkrete :-)

Insbesondere gilt:

$$[\![\pi]\!]\, s \in \gamma\,([\![\pi]\!]^\sharp\, D)$$

Die abstrakte Semantik simuliert die konkrete :-)

Insbesondere gilt:

$$[\![\pi]\!]\, s \in \gamma\,([\![\pi]\!]^\sharp\, D)$$

Praktisch heißt das z.B., dass für $D\,x = -7$ gilt:

$$\begin{array}{lrcll} & \rho'\,x & = & -7 & \text{für alle} \quad \rho' \in \gamma\,D \\ \Longrightarrow & \rho_1\,x & = & -7 & \text{für} \quad (\rho_1, \_) = [\![\pi]\!]\, s \end{array}$$

Zum Beweis von $(*)$ zeigen wir für jede Kante k :

$$(**) \qquad \begin{tikzcd} \boxed{s} \arrow[r, "[\![k]\!]"] \arrow[d, dash, "\Delta"'] & \boxed{s_1} \arrow[d, dash, dotted, "\Delta"] \\ \boxed{D} \arrow[r, "[\![k]\!]^\sharp"'] & \boxed{D_1} \end{tikzcd}$$

Dann folgt $(*)$ mittels Induktion :-)

Zum Beweis von $(**)$ zeigen wir für jeden Ausdruck e :

$(***)$ $\quad (\llbracket e \rrbracket\, \rho) \;\Delta\; (\llbracket e \rrbracket^\sharp\, D)$ sofern nur $\rho \,\Delta\, D$

Zum Beweis von $(**)$ zeigen wir für jeden Ausdruck e :

$(***)$ $\quad ([\![e]\!]\,\rho) \;\Delta\; ([\![e]\!]^\sharp\, D)$ sofern nur $\rho \,\Delta\, D$

Zum Beweis von $(***)$ zeigen wir für jeden Operator $\Box$:

$$(x \,\Box\, y) \;\Delta\; (x^\sharp \,\Box^\sharp\, y^\sharp) \qquad \text{sofern} \quad x \,\Delta\, x^\sharp \wedge y \,\Delta\, y^\sharp$$

Zum Beweis von $(**)$ zeigen wir für jeden Ausdruck e :

$(***)$ $\quad (\llbracket e \rrbracket\, \rho) \;\Delta\; (\llbracket e \rrbracket^\sharp\, D)$ sofern nur $\rho \,\Delta\, D$

Zum Beweis von $(***)$ zeigen wir für jeden Operator $\Box$:

$$(x \Box y) \;\Delta\; (x^\sharp \Box^\sharp y^\sharp) \qquad \text{sofern} \quad x \,\Delta\, x^\sharp \wedge y \,\Delta\, y^\sharp$$

So hatten wir die Operatoren $\Box^\sharp$ aber gerade definiert :-)

Nun zeigen wir $(**)$ durch Fallunterscheidung nach der Kanten-Beschriftung *lab* .

Sei $s = (\rho, \mu) \;\Delta\; D$. Insbesondere ist $\bot \neq D \;:\; \mathit{Vars} \to \mathbb{Z}^\top$

Fall $\boxed{x = e;}$:

$$\begin{array}{lll} \rho_1 & = & \rho \oplus \{x \mapsto [\![e]\!]\,\rho\} \qquad \mu_1 \;=\; \mu \\ D_1 & = & D \oplus \{x \mapsto [\![e]\!]^\sharp\, D\} \end{array}$$

$$\Longrightarrow \qquad (\rho_1, \mu_1) \;\Delta\; D_1$$

Fall $\boxed{x = M[R];}$:

$$\rho_1 = \rho \oplus \{x \mapsto \mu\,(\rho\, R)\} \qquad \mu_1 = \mu$$

$$D_1 = D \oplus \{x \mapsto \top\}$$

$$\Longrightarrow \qquad (\rho_1, \mu_1)\ \Delta\ D_1$$

Fall $\boxed{M[R_1] = R_2;}$:

$$\rho_1 = \rho \qquad \mu_1 = \mu \oplus \{\rho\, R_1 \mapsto \rho\, R_2\}$$

$$D_1 = D$$

$$\Longrightarrow \qquad (\rho_1, \mu_1)\ \Delta\ D_1$$

Fall $\boxed{\text{Neg}(e)}$ **:** $(\rho_1, \mu_1) = s\,,$ wobei:

$$
\begin{array}{rccll}
 & 0 & = & [\![e]\!]\,\rho & \\
 & & \Delta & [\![e]\!]^\sharp\,D & \\
\Longrightarrow & 0 & \sqsubseteq & [\![e]\!]^\sharp\,D & \\
\Longrightarrow & \bot & \neq & D_1 & = \; D \\
\Longrightarrow & \multicolumn{4}{l}{(\rho_1, \mu_1)\;\Delta\;D_1}
\end{array}
$$

Fall $\boxed{\mathrm{Pos}(e)}$: $(\rho_1, \mu_1) = s\,,$ wobei:

$$\begin{array}{rlll} & 0 & \neq & [\![e]\!]\,\rho \\ & & \Delta & [\![e]\!]^\sharp\,D \\ \Longrightarrow & 0 & \neq & [\![e]\!]^\sharp\,D \\ \Longrightarrow & \bot & \neq & D_1 \;=\; D \\ \Longrightarrow & (\rho_1, \mu_1)\;\Delta\;D_1 & & \end{array}$$

:-)

Wir schließen: Die Behauptung $(*)$ stimmt :-))

Die MOP-Lösung:

$$\mathcal{D}^*[v] = \bigsqcup\{[\![\pi]\!]^\sharp\, D_0 \mid \pi : start \to^* v\}$$

wobei $D_0\, x = \top \qquad (x \in Vars)$.

Wir schließen: Die Behauptung $(*)$ stimmt :-))

Die MOP-Lösung:

$$\mathcal{D}^*[v] \;=\; \bigsqcup\{[\![\pi]\!]^\sharp\, D_0 \mid \pi : \mathit{start} \rightarrow^* v\}$$

wobei $D_0\, x = \top \qquad (x \in \mathit{Vars})$.

Wegen $(*)$ gilt für alle Anfangszustände s und alle Berechnungen π , die v erreichen:

$$([\![\pi]\!]\, s) \;\;\Delta\;\; (\mathcal{D}^*[v])$$

Wir schließen: Die Behauptung $(*)$ stimmt :-))

Die MOP-Lösung:

$$\mathcal{D}^*[v] = \bigsqcup \{ [\![\pi]\!]^\sharp \, D_0 \mid \pi : \mathit{start} \rightarrow^* v \}$$

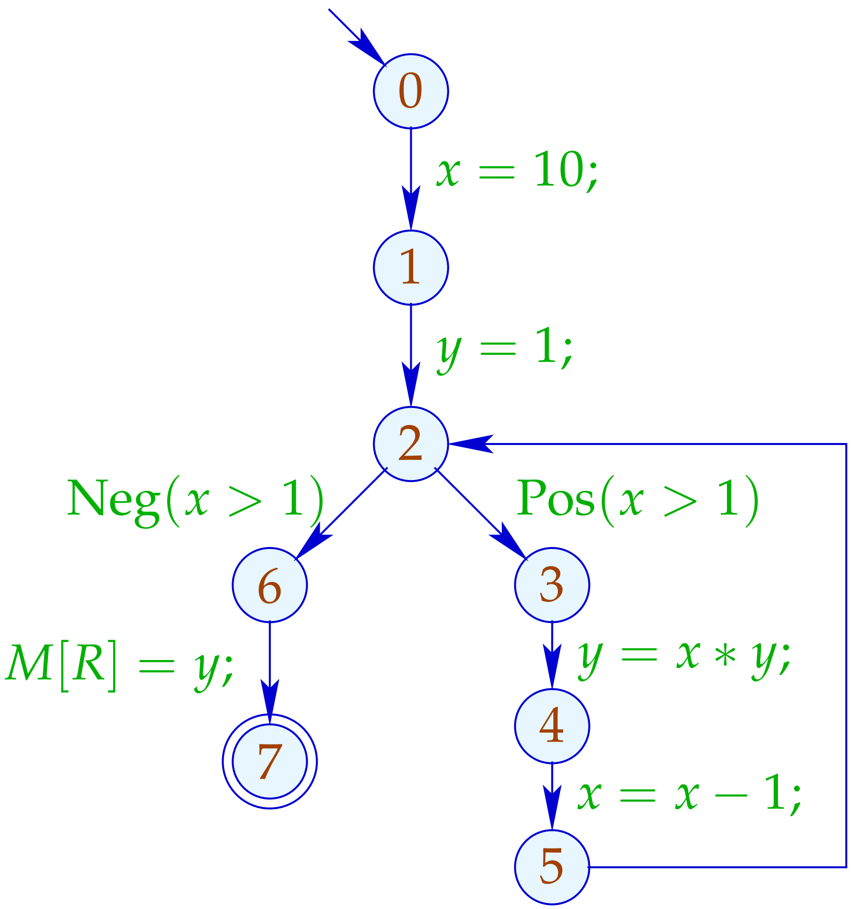
wobei $\quad D_0 \, x = \top \quad (x \in \mathit{Vars})$.

Wegen $(*)$ gilt für alle Anfangszustände s und alle Berechnungen π , die v erreichen:

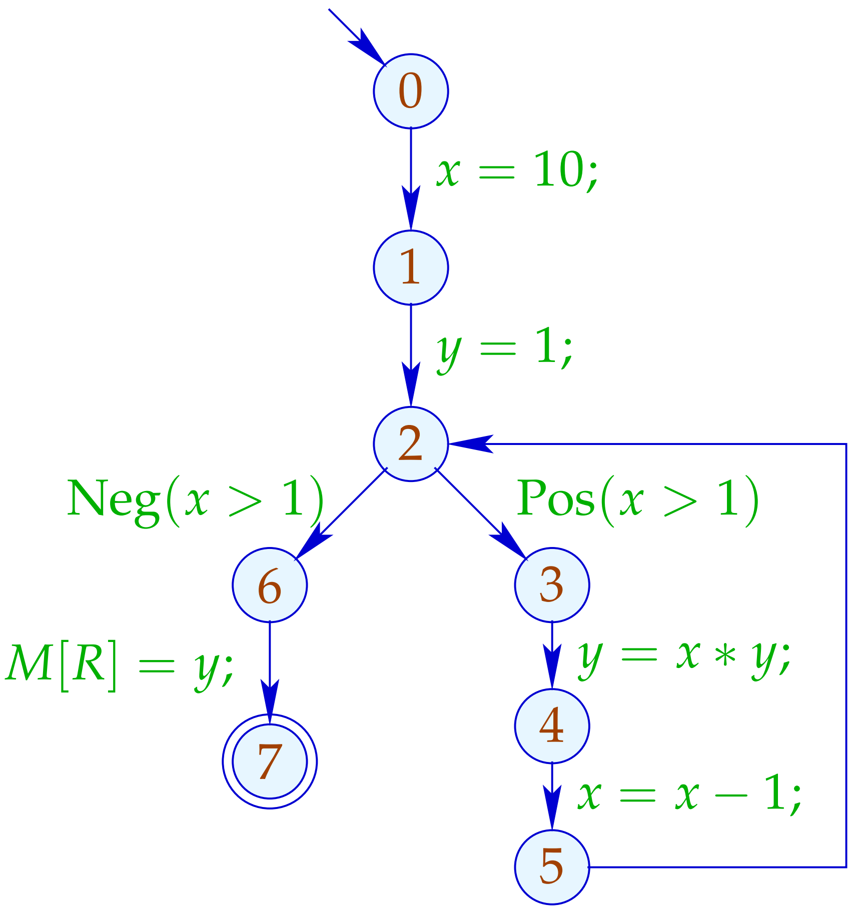
$$([\![\pi]\!] \, s) \;\; \Delta \;\; (\mathcal{D}^*[v])$$

Zur Approximation des MOP benutzen wir unser Constraint-System :-))

Beispiel:

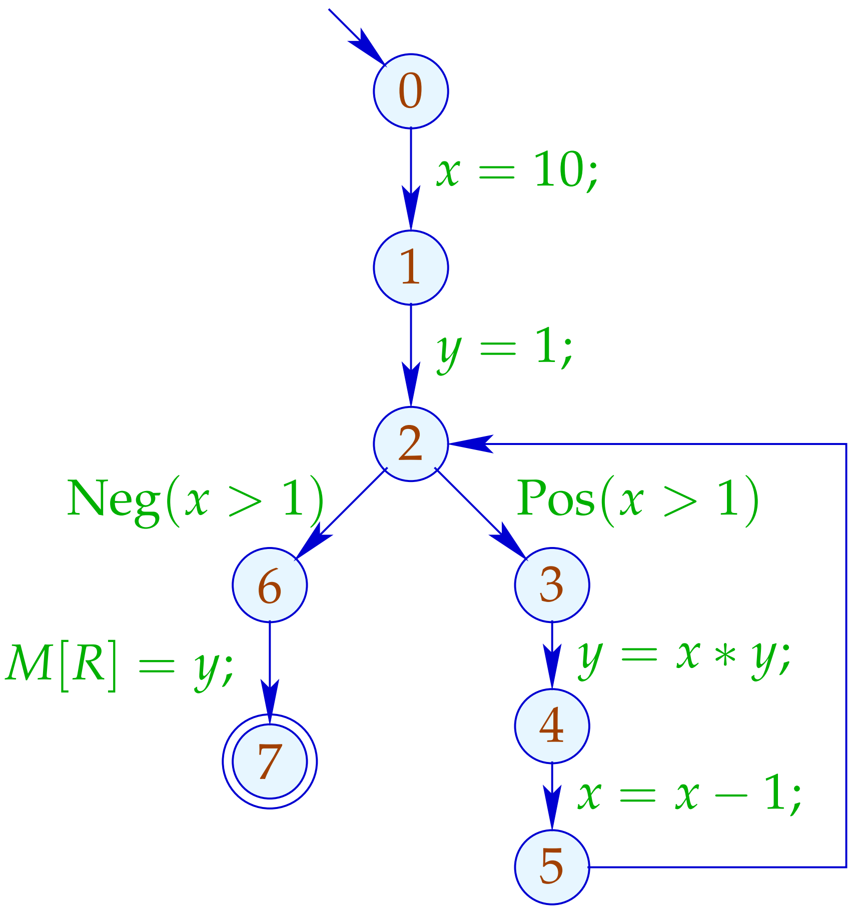


Beispiel:



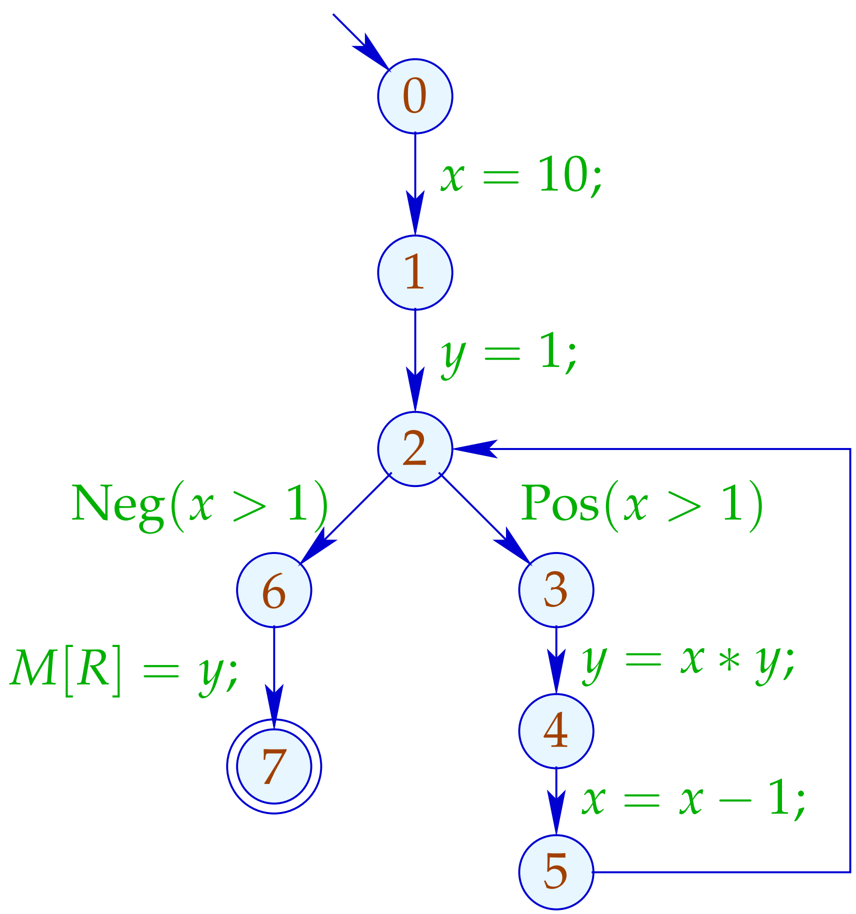
	1	
	x	y
0	$\top$	$\top$
1	10	$\top$
2	10	1
3	10	1
4	10	10
5	9	10
6	$\bot$	
7	$\bot$	

Beispiel:



	1		2	
	x	y	x	y
0	$\top$	$\top$	$\top$	$\top$
1	10	$\top$	10	$\top$
2	10	1	$\top$	$\top$
3	10	1	$\top$	$\top$
4	10	10	$\top$	$\top$
5	9	10	$\top$	$\top$
6	$\perp$		$\top$	$\top$
7	$\perp$		$\top$	$\top$

Beispiel:



	1		2		3	
	x	y	x	y	x	y
0	$\top$	$\top$	$\top$	$\top$	dito	
1	10	$\top$	10	$\top$		
2	10	1	$\top$	$\top$		
3	10	1	$\top$	$\top$		
4	10	10	$\top$	$\top$		
5	9	10	$\top$	$\top$		
6	$\bot$		$\top$	$\top$		
7	$\bot$		$\top$	$\top$		

Fazit:

Obwohl wir mit konkreten Zahlen rechnen, kriegen wir nicht alles raus :-(

Dafür terminiert die Fixpunkt-Iteration garantiert:

Für n Programmpunkte und m Variablen benötigen wir maximal: $n \cdot (m+1)$ Runden :-)

Achtung:

Die Kanten-Effekte sind nicht distributiv !!!

Gegenbeispiel: $f = [\![x = x + y;]\!]^\sharp$

$$\begin{array}{llll}
\textbf{Sei} & D_1 & = & \{x \mapsto 2, y \mapsto 3\} \\
 & D_2 & = & \{x \mapsto 3, y \mapsto 2\} \\
\textbf{Dann} & f\, D_1 \sqcup f\, D_2 & = & \{x \mapsto 5, y \mapsto 3\} \sqcup \{x \mapsto 5, y \mapsto 2\} \\
 & & = & \{x \mapsto 5, y \mapsto \top\} \\
 & & \neq & \{x \mapsto \top, y \mapsto \top\} \\
 & & = & f\, \{x \mapsto \top, y \mapsto \top\} \\
 & & = & f\, (D_1 \sqcup D_2)
\end{array}$$

:-((

Wir schließen:

Die kleinste Lösung $\mathcal{D}$ des Constraint-Systems liefert i.a. nur eine obere Approximation des MOP, d.h.:

$$\mathcal{D}^*[v] \quad \sqsubseteq \quad \mathcal{D}[v]$$

Wir schließen:

Die kleinste Lösung $\mathcal{D}$ des Constraint-Systems liefert i.a. nur eine obere Approximation des MOP, d.h.:

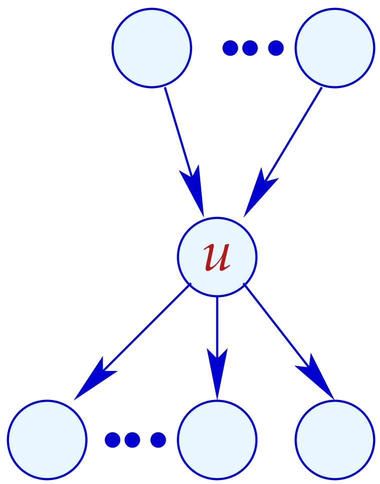
$$\mathcal{D}^*[v] \quad \sqsubseteq \quad \mathcal{D}[v]$$

Als obere Approximation beschreibt $\mathcal{D}[v]$ trotzdem das Ergebnis jeder Berechnung π , die in v endet:

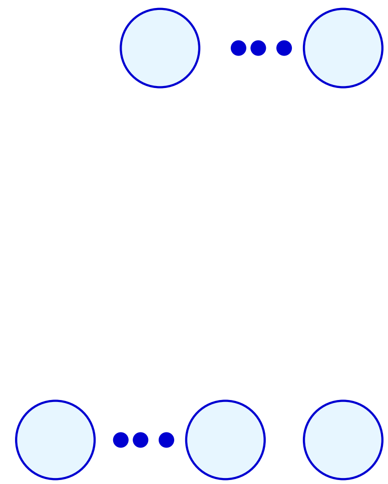
$$([\![\pi]\!]\,(\rho, \mu)) \;\; \Delta \;\; (\mathcal{D}[v])$$

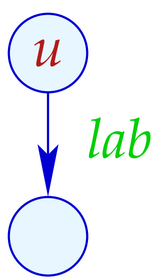
wann immer $[\![\pi]\!]\,(\rho, \mu)$ definiert ist ;-))

Transformation 5: Beseitigung von totem Code

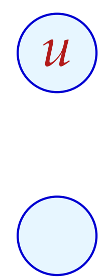


$$\xrightarrow{\mathcal{D}[u] = \bot}$$





$$\xrightarrow{[\![lab]\!]^\sharp(\mathcal{D}[u]) = \bot}$$



Transformation 5 (Forts.): Beseitigung von totem Code

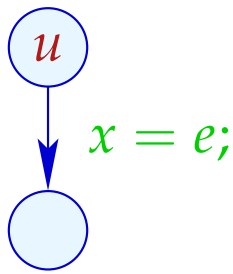
$$\perp \neq \mathcal{D}[u] = D \qquad [\![e]\!]^\sharp\, D = 0$$

$$u \xrightarrow{\text{Neg}\ (e)} \bigcirc \quad \Longrightarrow \quad u \xrightarrow{\ ;\ } \bigcirc$$

$$\perp \neq \mathcal{D}[u] = D \qquad [\![e]\!]^\sharp\, D \not\in \{0, \top\}$$

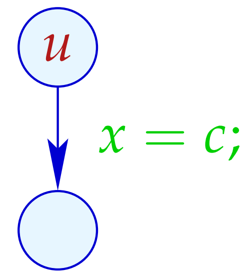
$$u \xrightarrow{\text{Pos}\ (e)} \bigcirc \quad \Longrightarrow \quad u \xrightarrow{\ ;\ } \bigcirc$$

Transformation 5 (Forts.): Vereinfachte Zuweisungen



$$\bot \neq \mathcal{D}[u] = D$$
$$[\![e]\!]^\sharp\, D = c$$

$$\Longrightarrow$$



Erweiterungen:

- Statt ganzer rechter Seiten kann man auch Teilausdrücke vereinfachen:

$$x + (3 * y) \quad \xrightarrow{\{x \mapsto \top, y \mapsto 5\}} \quad x + 15$$

... und weitere Vereinfachungsregeln anwenden, etwa:

$$\begin{array}{rcl} x * 0 & \Longrightarrow & 0 \\ x * 1 & \Longrightarrow & x \\ x + 0 & \Longrightarrow & x \\ x - 0 & \Longrightarrow & x \\ & \ldots & \end{array}$$

- Bisher haben wir die Information von Bedingungen nicht optimal ausgenutzt:

$$\text{if } (x == 7)$$
$$y = x + 3;$$

Selbst wenn wir den Wert von x vor der if-Abfrage nicht kennen, wissen wir doch, dass bei Betreten des then-Teils x stets den Wert 7 hat :-)

Wir könnten darum definieren:

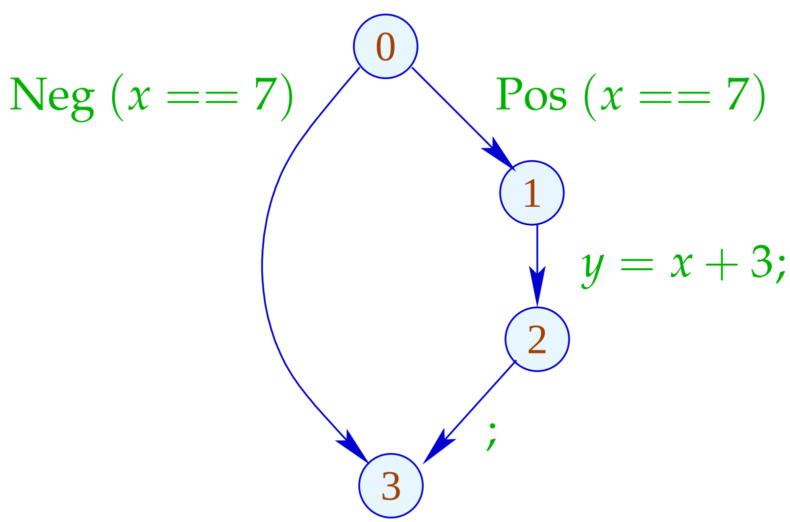
$$[\![\text{Pos}\,(x == e)]\!]^\sharp\, D = \begin{cases} D & \text{falls} \quad [\![x == e]\!]^\sharp\, D = 1 \\ \bot & \text{falls} \quad [\![x == e]\!]^\sharp\, D = 0 \\ D_1 & \text{sonst} \end{cases}$$

wobei

$$D_1 = D \oplus \{x \mapsto (D\,x \sqcap [\![e]\!]^\sharp\, D)\}$$

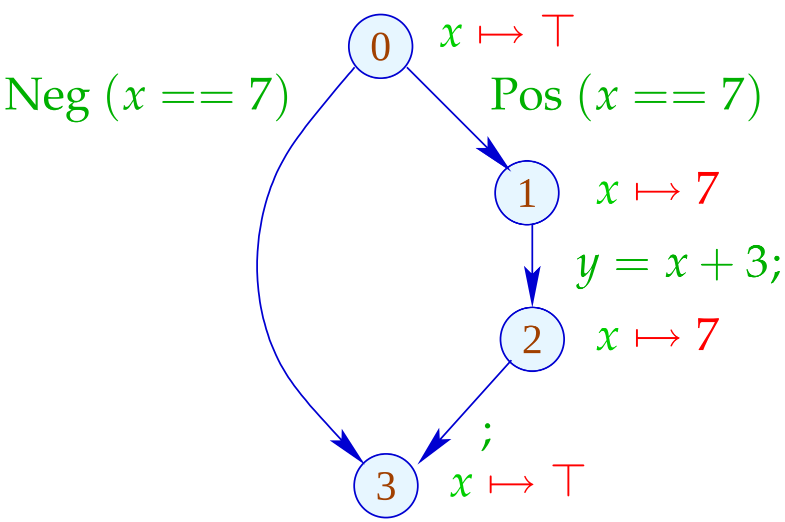
Analog sieht der Kanteneffekt für $\text{Neg}\,(x \neq e)$ aus :-)

Unser Beispiel:



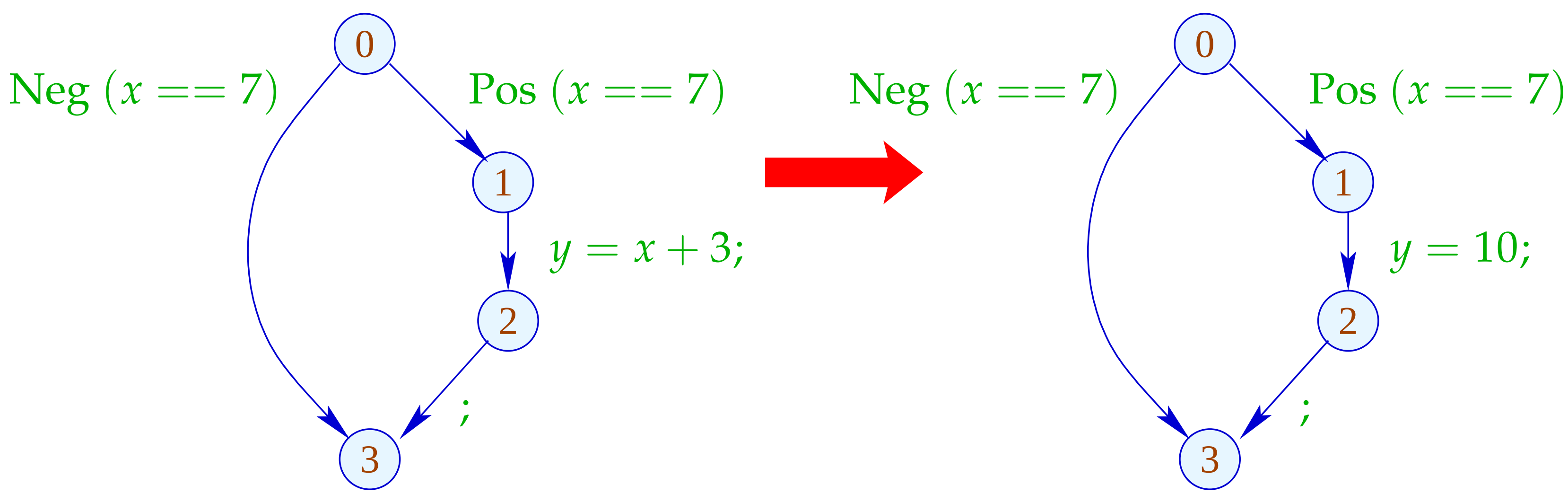
Analog sieht der Kanteneffekt für $\text{Neg}\,(x \neq e)$ aus :-)

Unser Beispiel:



Analog sieht der Kanteneffekt für $\mathrm{Neg}\,(x \neq e)$ aus :-)

Unser Beispiel:



1.5 Intervall-Analyse

Beobachtung:

- Programmiererinnen benutzen oft globale Konstanten, um Debug-Code ein oder aus zu schalten

  $\Longrightarrow$

  Konstantenpropagation ist hilfreich :-)

- Im allgemeinen wird aber der Wert von Variablen nicht bekannt sein — möglicherweise aber ein Intervall !!!

Beispiel:

$$\text{for } (i = 0; i < 42; i{+}{+})$$
$$\quad \text{if } (0 \leq i \wedge i < 42)\{$$
$$\qquad A_1 = A + i;$$
$$\qquad M[A_1] = i;$$
$$\quad \}$$

// A Anfangsadresse eines Felds

// if ist Array-Bound-Check

Offenbar ist die innere Abfrage überflüssig :-)

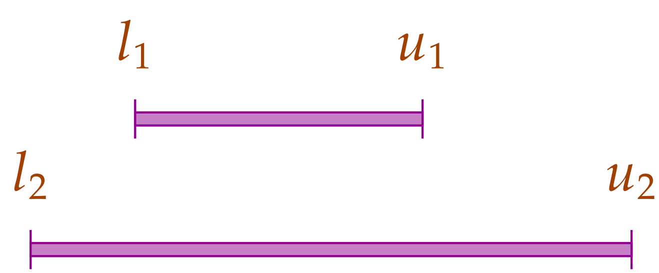
Idee 1:

Bestimme für jede Variable x ein (möglichst kleines :-) Intervall für die möglichen Werte:

$$\mathbb{I} = \{[l, u] \mid l \in \mathbb{Z} \cup \{-\infty\}, u \in \mathbb{Z} \cup \{+\infty\}, l \leq u\}$$

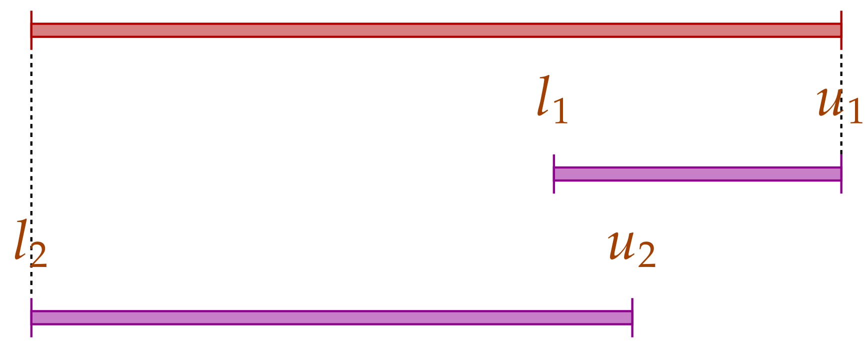
Partielle Ordnung:

$$[l_1, u_1] \sqsubseteq [l_2, u_2] \qquad \text{gdw.} \qquad l_2 \leq l_1 \wedge u_1 \leq u_2$$



Damit:

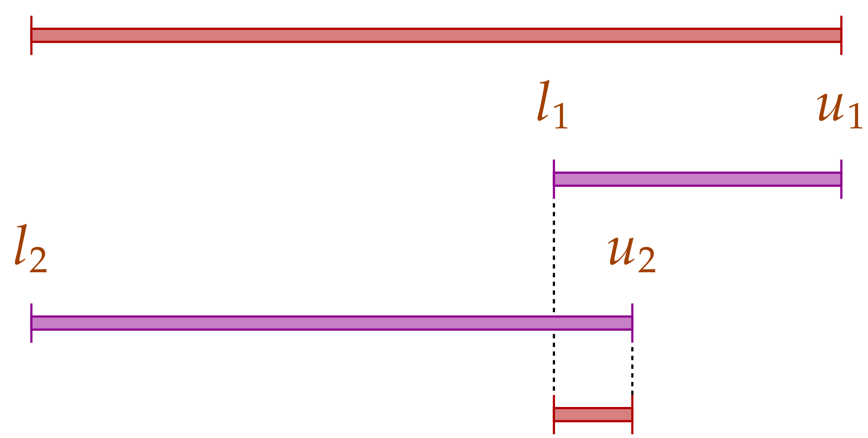
$$[l_1, u_1] \sqcup [l_2, u_2] \quad = \qquad [l_1 \sqcap l_2, u_1 \sqcup u_2]$$



Damit:

$$[l_1, u_1] \sqcup [l_2, u_2] \quad = \quad [l_1 \sqcap l_2, u_1 \sqcup u_2]$$

$$[l_1, u_1] \sqcap [l_2, u_2] \quad = \quad [l_1 \sqcup l_2, u_1 \sqcap u_2] \qquad \text{sofern } (l_1 \sqcup l_2) \leq (u_1 \sqcap u_2)$$



Achtung:

$\rightarrow$ $\mathbb{I}$ ist kein vollständiger Verband :-)

$\rightarrow$ $\mathbb{I}$ besitzt unendliche aufsteigende Ketten, z.B.

$$[0, 0] \sqsubset [0, 1] \sqsubset [-1, 1] \sqsubset [-1, 2] \sqsubset \ldots$$

Achtung:

$\rightarrow$ $\mathbb{I}$ ist kein vollständiger Verband :-)

$\rightarrow$ $\mathbb{I}$ besitzt unendliche aufsteigende Ketten, z.B.

$$[0,0] \sqsubset [0,1] \sqsubset [-1,1] \sqsubset [-1,2] \sqsubset \ldots$$

Beschreibungsrelation:

$$z \;\Delta\; [l,u] \qquad \text{gdw.} \qquad l \leq z \leq u$$

Konkretisierung:

$$\gamma\,[l,u] = \{z \in \mathbb{Z} \mid l \leq z \leq u\}$$

Beispiel:

$$\begin{array}{lll} \gamma\,[0,7] & = & \{0,\ldots,7\} \\ \gamma\,[0,\infty] & = & \{0,1,2,\ldots,\} \end{array}$$

Rechnen mit Intervallen:

Intervall-Arithmetik :-)

Addition:

$$\begin{array}{rcl} [l_1,u_1] +^\sharp [l_2,u_2] & = & [l_1+l_2, u_1+u_2] \qquad \text{wobei} \\ -\infty + \_ & = & -\infty \\ +\infty + \_ & = & +\infty \\ & // & -\infty + \infty \quad \text{kommt nicht vor} \quad :\text{-}) \end{array}$$

Negation:

$$-^\sharp [l,u] \quad = \quad [-u,-l]$$

Multiplikation:

$$\begin{array}{rcll} [l_1,u_1] \;*^\sharp\; [l_2,u_2] & = & [a,b] & \qquad \text{wobei} \\ a & = & l_1 l_2 \sqcap l_1 u_2 \sqcap u_1 l_2 \sqcap u_1 u_2 & \\ b & = & l_1 l_2 \sqcup l_1 u_2 \sqcup u_1 l_2 \sqcup u_1 u_2 & \end{array}$$

Beispiel:

$$\begin{array}{rcl} [0,2] \;*^\sharp\; [3,4] & = & [0,8] \\ [-1,2] \;*^\sharp\; [3,4] & = & [-4,8] \\ [-1,2] \;*^\sharp\; [-3,4] & = & [-6,8] \\ [-1,2] \;*^\sharp\; [-4,-3] & = & [-8,4] \end{array}$$

Division: $[l_1, u_1] \,/^\sharp\, [l_2, u_2] \;=\; [a, b]$

- Ist 0 nicht im Nenner-Intervall enthalten, sei:

$$\begin{array}{lll} a & = & l_1/l_2 \sqcap l_1/u_2 \sqcap u_1/l_2 \sqcap u_1/u_2 \\ b & = & l_1/l_2 \sqcup l_1/u_2 \sqcup u_1/l_2 \sqcup u_1/u_2 \end{array}$$

- Gilt: $l_2 \leq 0 \leq u_2$, setzen wir:

$$[a, b] \;=\; [-\infty, +\infty]$$

Gleichheit:

$$[l_1, u_1] ==^\sharp [l_2, u_2] \;=\; \begin{cases} [1,1] & \text{falls} \quad l_1 = u_1 = l_2 = u_2 \\ [0,0] & \text{falls} \quad u_1 < l_2 \vee u_2 < l_1 \\ [0,1] & \text{sonst} \end{cases}$$

Gleichheit:

$$[l_1, u_1] ==^\sharp [l_2, u_2] \;=\; \begin{cases} [1,1] & \text{falls} \quad l_1 = u_1 = l_2 = u_2 \\ [0,0] & \text{falls} \quad u_1 < l_2 \vee u_2 < l_1 \\ [0,1] & \text{sonst} \end{cases}$$

Beispiel:

$$\begin{aligned} [42,42] ==^\sharp [42,42] &= [1,1] \\ [0,7] ==^\sharp [0,7] &= [0,1] \\ [1,2] ==^\sharp [3,4] &= [0,0] \end{aligned}$$

Kleiner:

$$[l_1, u_1] <^\sharp [l_2, u_2] = \begin{cases} [1,1] & \text{falls} \quad u_1 < l_2 \\ [0,0] & \text{falls} \quad u_2 \leq l_1 \\ [0,1] & \text{sonst} \end{cases}$$

Kleiner:

$$[l_1, u_1] \;<^\sharp\; [l_2, u_2] \;=\; \begin{cases} [1,1] & \text{falls} \quad u_1 < l_2 \\ [0,0] & \text{falls} \quad u_2 \leq l_1 \\ [0,1] & \text{sonst} \end{cases}$$

Beispiel:

$$\begin{aligned} [1,2] <^\sharp [9,42] &= [1,1] \\ [0,7] \;<^\sharp\; [0,7] &= [0,1] \\ [3,4] \;<^\sharp\; [1,2] &= [0,0] \end{aligned}$$

Mithilfe von $\mathbb{I}$ konstruieren wir den vollständigen Verband:

$$\mathbb{D}_{\mathbb{I}} = (Vars \to \mathbb{I})_\perp$$

Beschreibungsrelation:

$$\rho \;\Delta\; D \qquad \text{gdw.} \qquad D \neq \perp \quad \wedge \quad \forall\, x \in Vars : \;(\rho\, x)\;\Delta\;(D\; x)$$

Die abstrakte Ausdrucksauswertung definieren wir analog Konstantenpropagation. Wir finden:

$$([\![e]\!]\;\rho)\;\;\Delta\;\;([\![e]\!]^\sharp\;D) \qquad \text{sofern} \qquad \rho\;\Delta\;D$$

Die Kanteneffekte:

$$
\begin{array}{lll}
[\![;]\!]^\sharp\, D & = & D \\
[\![x = e;]\!]^\sharp\, D & = & D \oplus \{x \mapsto [\![e]\!]^\sharp\, D\} \\
[\![x = M[R];]\!]^\sharp\, D & = & D \oplus \{x \mapsto \top\} \\
[\![M[R_1] = R_2;]\!]^\sharp\, D & = & D \\
[\![\mathrm{Pos}\,(e)]\!]^\sharp\, D & = & \begin{cases} \bot & \text{falls} \quad [0,0] = [\![e]\!]^\sharp\, D \\ D & \text{sonst} \end{cases} \\
[\![\mathrm{Neg}\,(e)]\!]^\sharp\, D & = & \begin{cases} D & \text{falls} \quad [0,0] \sqsubseteq [\![e]\!]^\sharp\, D \\ \bot & \text{sonst} \end{cases}
\end{array}
$$

... sofern $\quad D \neq \bot \quad$:-)

Bessere Ausnutzung von Bedingungen:

$$[\![\mathrm{Pos}\,(e)]\!]^\sharp\,D \;=\; \begin{cases} \bot & \text{falls} \quad [0,0] = [\![e]\!]^\sharp\,D \\ D_1 & \text{sonst} \end{cases}$$

wobei :

$$D_1 \;=\; \begin{cases} D \oplus \{x \mapsto (D\,x) \sqcap ([\![e_1]\!]^\sharp\,D)\} & \text{falls } e \equiv x == e_1 \\ D \oplus \{x \mapsto (D\,x) \sqcap [-\infty, u]\} & \text{falls } e \equiv x \;\leq\; e_1, [\![e_1]\!]^\sharp\,D = [\_, u] \\ D \oplus \{x \mapsto (D\,x) \sqcap [l, \infty]\} & \text{falls } e \equiv x \;\geq\; e_1, [\![e_1]\!]^\sharp\,D = [l, \_] \end{cases}$$

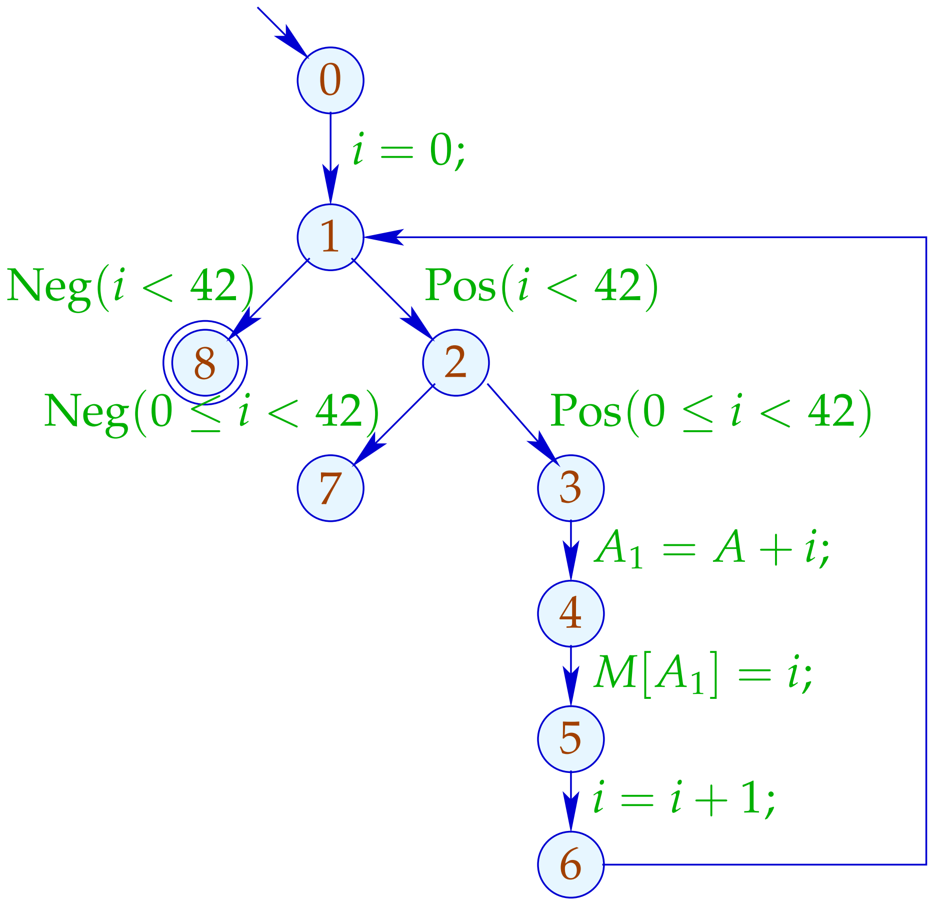
Bessere Ausnutzung von Bedingungen (Forts.):

$$[\![\mathrm{Neg}\,(e)]\!]^\sharp\,D \quad = \quad \begin{cases} \bot & \text{falls} \quad [0,0] \not\sqsubseteq [\![e]\!]^\sharp\,D \\ D_1 & \text{sonst} \end{cases}$$

wobei :

$$D_1 \quad = \quad \begin{cases} D \oplus \{x \mapsto (D\,x) \sqcap ([\![e_1]\!]^\sharp\,D)\} & \text{falls } e \equiv x \neq e_1 \\ D \oplus \{x \mapsto (D\,x) \sqcap [-\infty, u]\} & \text{falls } e \equiv x > e_1, [\![e_1]\!]^\sharp\,D = [\_, u] \\ D \oplus \{x \mapsto (D\,x) \sqcap [l, \infty]\} & \text{falls } e \equiv x < e_1, [\![e_1]\!]^\sharp\,D = [l, \_] \end{cases}$$

Beispiel:



	i	
	l	u
0	$-\infty$	$+\infty$
1	0	42
2	0	41
3	0	41
4	0	41
5	0	41
6	1	42
7	$\perp$	
8	42	42

Problem:

$\rightarrow$ Die Lösung lässt sich mit RR-Iteration berechnen — nach ca. 42 Runden :-(

$\rightarrow$ Auf manchen Programmen terminiert die Iteration nie :-((

Idee 1: Widening

- Iteriere beschleunigt — unter Preisgabe von Präzision :-)
- Erlaube nur beschränkt oft die Modifikation eines Werts !!!

... im Beispiel:

- verbiete Updates von Intervall-Grenzen in $\mathbb{Z}$ ...

$\Longrightarrow$ eine maximale Kette:

$$[3, 17] \sqsubset [3, +\infty] \sqsubset [-\infty, +\infty]$$

Formalisierung dieses Vorgehens:

Sei $x_i \sqsupseteq f_i(x_1, \ldots, x_n)\,, \quad i = 1, \ldots, n$ (1)

ein Ungleichungsssystem über $\mathbb{D}$, wobei die f_i nicht notwendigerweise monoton sind.

Trotzdem können wir eine akkumulierende Iteration definieren. Betrachte das Gleichungssystem:

$$x_i = x_i \sqcup f_i(x_1, \ldots, x_n)\,, \quad i = 1, \ldots, n \qquad (2)$$

Offenbar gilt:

(a) $\underline{x}$ ist Lösung von (1) gdw. $\underline{x}$ Lösung von (2) ist.

(b) Die Funktion $G : \mathbb{D}^n \to \mathbb{D}^n$ mit
$G(x_1, \ldots, x_n) = (y_1, \ldots, y_n)\,, \quad y_i = x_i \sqcup f_i(x_1, \ldots, x_n)$
ist vergrößernd, d.h. $\underline{x} \sqsubseteq G\,\underline{x}$ für alle $\underline{x} \in \mathbb{D}^n$.

(c) Die Folge $G^k \underline{\perp}$, $k \geq 0$, ist eine aufsteigende Kette:

$$\underline{\perp} \sqsubseteq G\underline{\perp} \sqsubseteq \ldots \sqsubseteq G^k\underline{\perp} \sqsubseteq \ldots$$

(d) Gilt $G^k \underline{\perp} = G^{k+1} \underline{\perp} = \underline{y}$ ist $\underline{y}$ eine Lösung von (1).

(e) Hat $\mathbb{D}$ unendliche aufsteigende Ketten, ist uns mit (d) noch nicht viel gedient ...

aber: wir könnten statt Gleichungssystem (2) ein Gleichungssystem:

$$x_i = x_i \sqcup\!\!\!\!\lrcorner\; f_i(x_1, \ldots, x_n)\,, \quad i = 1, \ldots, n \tag{3}$$

betrachten für eine binäre Operation Widening:

$$\sqcup\!\!\!\!\lrcorner : \mathbb{D}^2 \to \mathbb{D} \qquad \text{mit} \qquad v_1 \sqcup v_2 \sqsubseteq v_1 \sqcup\!\!\!\!\lrcorner\; v_2$$

Dann berechnet (RR)-Iteration für (3) immer noch eine Lösung von (1) :-)

... für die Intervall-Analyse:

- Der vollständige Verband ist: $\mathbb{D}_{\mathbb{I}} = (\mathit{Vars} \to \mathbb{I})_{\perp}$
- Das Widening $\sqcup\!\!\!\!\sqcup$ definieren wir als:

$$\perp \sqcup\!\!\!\!\sqcup D = D \sqcup\!\!\!\!\sqcup \perp = D \qquad \text{und für} \quad D_1 \neq \perp \neq D_2:$$

$$\begin{array}{rcl} (D_1 \sqcup\!\!\!\!\sqcup D_2)\, x & = & (D_1\, x) \sqcup\!\!\!\!\sqcup (D_2\, x) \qquad \text{wobei} \\ [l_1, u_1] \sqcup\!\!\!\!\sqcup [l_2, u_2] & = & [l, u] \qquad \text{mit} \\ l & = & \begin{cases} l_1 & \text{falls} \quad l_1 \leq l_2 \\ -\infty & \text{sonst} \end{cases} \\ u & = & \begin{cases} u_1 & \text{falls} \quad u_1 \geq u_2 \\ +\infty & \text{sonst} \end{cases} \end{array}$$

$\Longrightarrow$ $\sqcup\!\!\!\!\sqcup$ ist nicht kommutativ !!!

Beispiel:

$$
\begin{array}{lll}
[0,2] \sqcup [1,2] & = & [0,2] \\
[1,2] \sqcup [0,2] & = & [-\infty,2] \\
[1,5] \sqcup [3,7] & = & [1,+\infty]
\end{array}
$$

→ Widening liefert schneller größere Werte.

→ Es sollte so gewählt werden, dass es die Terminierung der Iteration garantiert :-)

→ Bei Intervall-Analyse begrenzt es die Anzahl der Iterationen auf:

$$\#\mathit{Punkte} \cdot (1 + 2 \cdot \#\mathit{Vars})$$

Fazit:

- Um eine Lösung von (1) über einem vollständigen Verband mit unendlichen aufsteigenden Ketten zu bestimmen, definieren wir ein geeignetes Widening und lösen dann (3) :-)

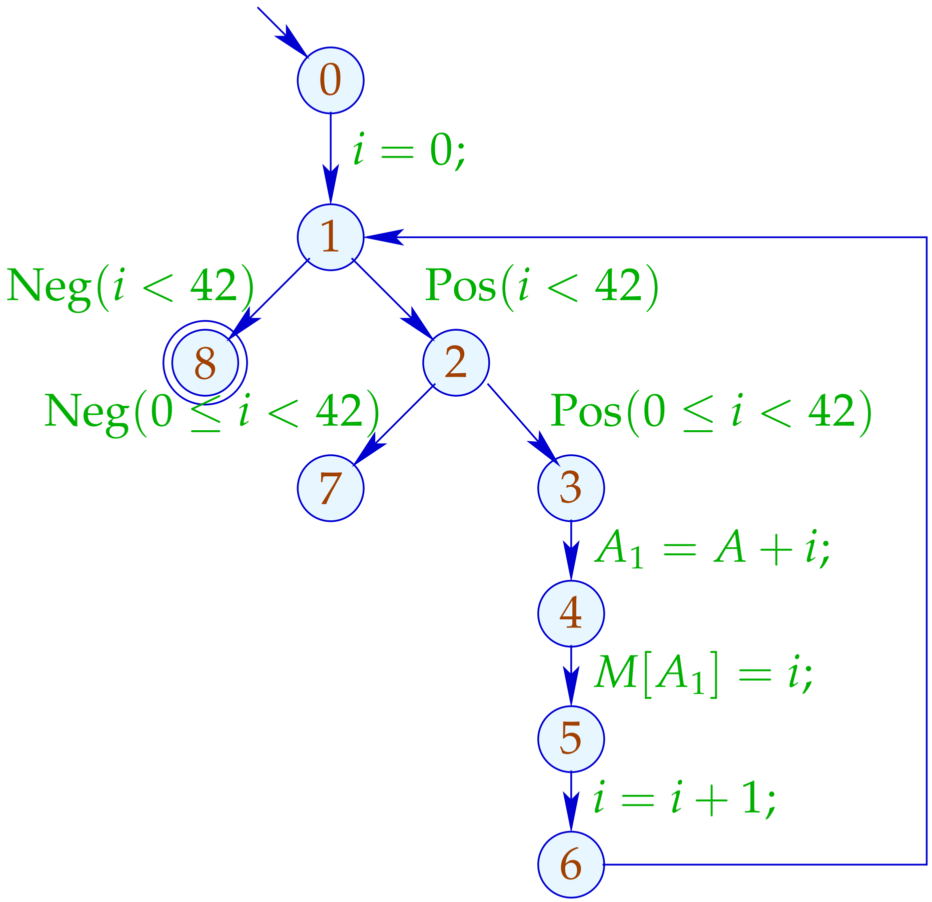
- Achtung: Die Konstruktion geeigneter Widenings ist eine schwarze Kunst !!!

  Oft wählt man $\sqcup\!\!\!\!\_$ ganz pragmatisch dynamisch während der Iteration, so dass

  $\longrightarrow$ die abstrakten Werte nicht zu kompliziert werden;

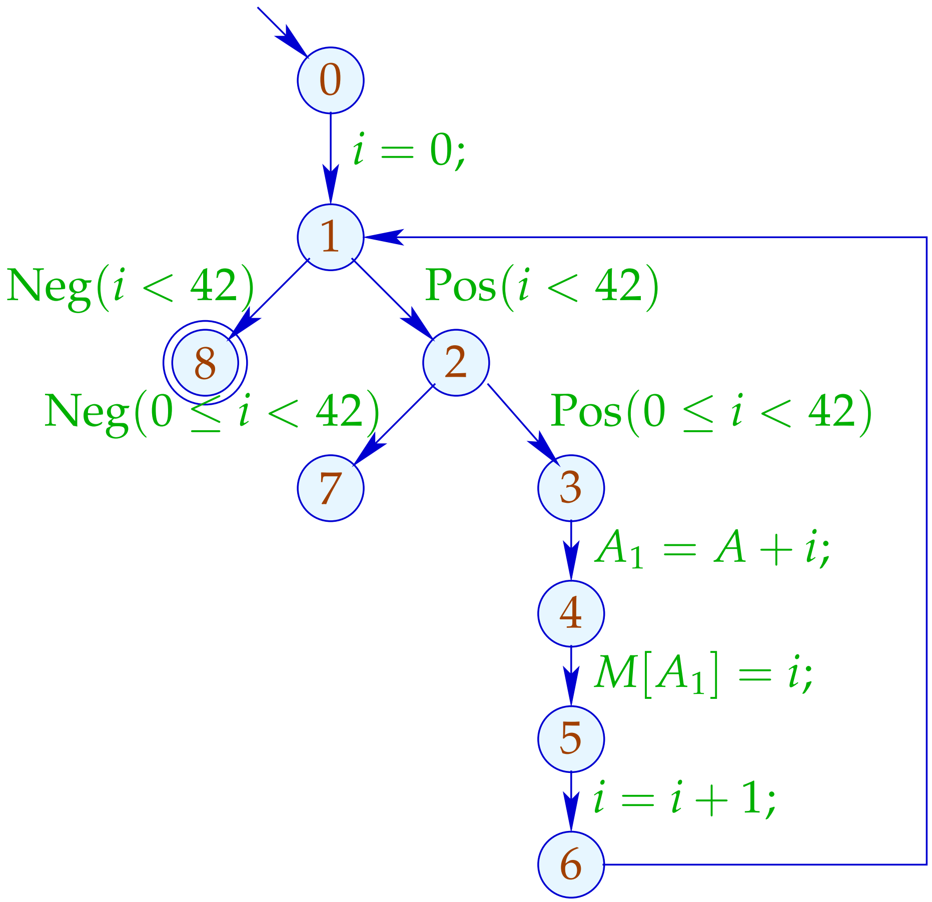
  $\longrightarrow$ die Anzahl der Updates fest beschränkt bleibt ...

Unser Beispiel:



	1	
	l	u
0	$-\infty$	$+\infty$
1	0	0
2	0	0
3	0	0
4	0	0
5	0	0
6	1	1
7	$\perp$	
8	$\perp$	

Unser Beispiel:



	1		2		3	
	l	u	l	u	l	u
0	$-\infty$	$+\infty$	$-\infty$	$+\infty$		
1	0	0	0	$+\infty$		
2	0	0	0	$+\infty$		
3	0	0	0	$+\infty$		
4	0	0	0	$+\infty$	dito	
5	0	0	0	$+\infty$		
6	1	1	1	$+\infty$		
7	$\perp$		42	$+\infty$		
8	$\perp$		42	$+\infty$		

... offenbar ist das Ergebnis enttäuschend :-(

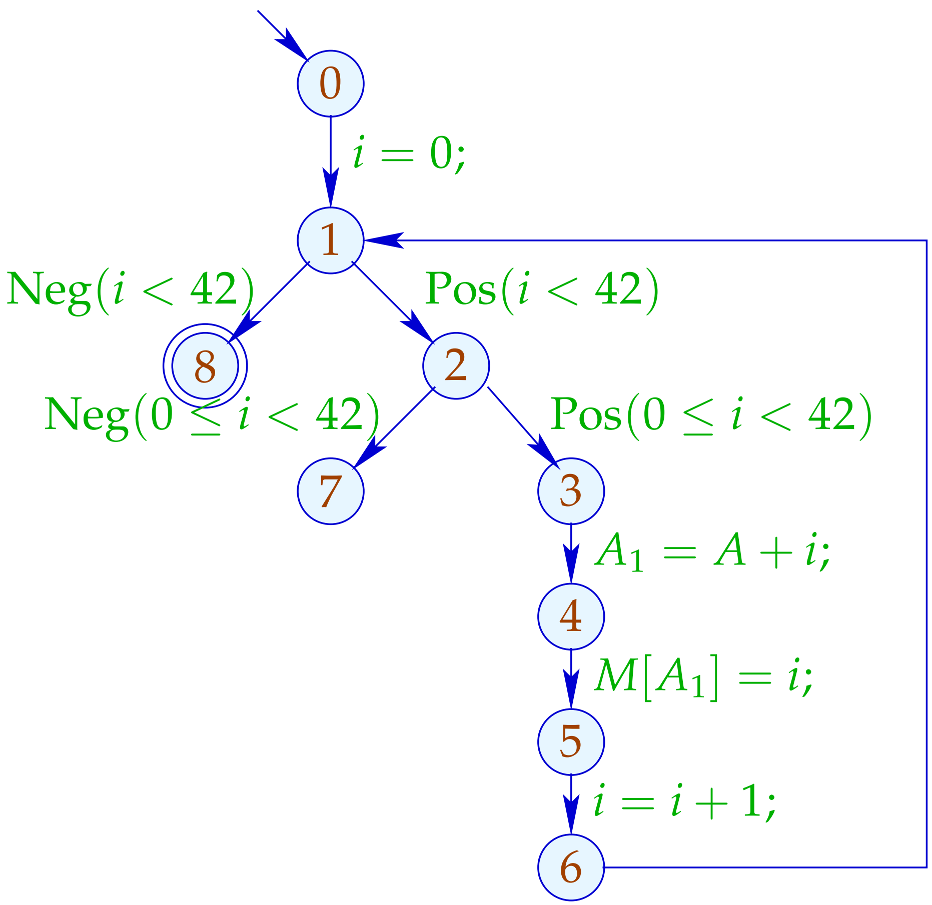
Idee 2:

Eigentlich reicht es, die Beschleunigung mittels $\sqcup$ nur an genügend vielen Stellen anzuwenden!

Eine Menge I heißt Loop Separator (Kreis-Trenner), falls jeder Kreis mindestens einen Punkt aus I enthält :-)

Wenden wir Widening nur an den Punkten aus einer solchen Menge I , terminiert RR-Iteration immer noch !!!

In unserem Beispiel:

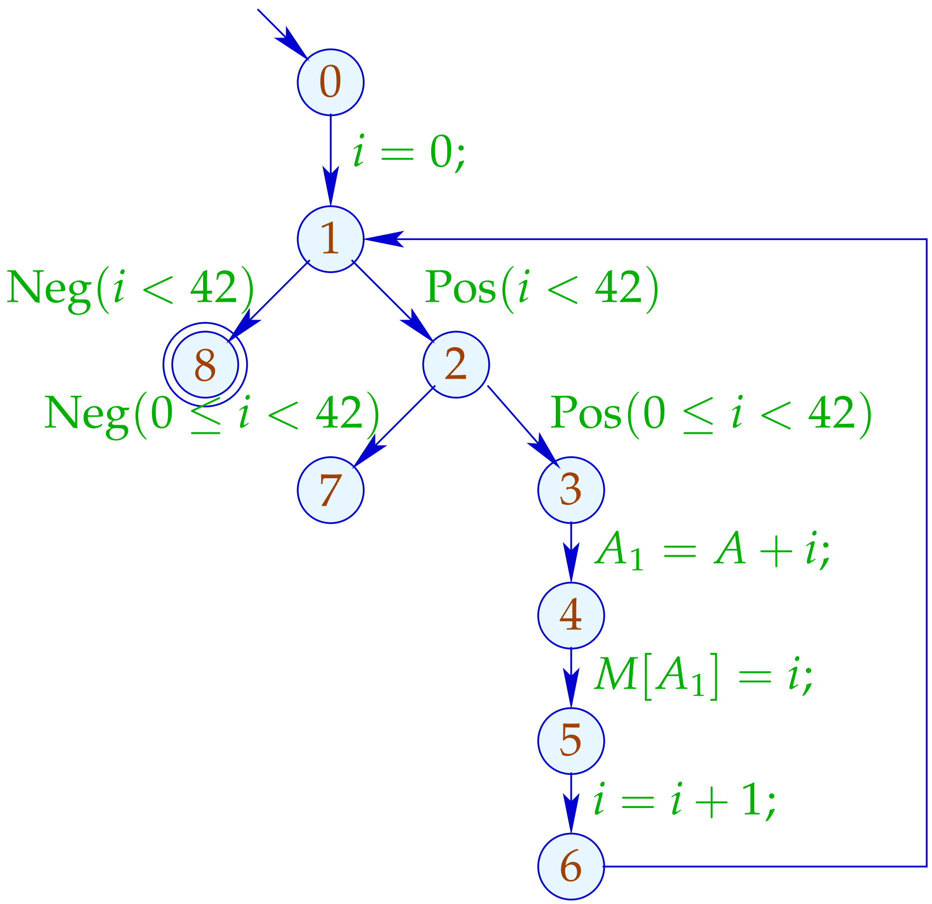


$I_1 \quad = \quad \{1\}$ oder auch:

$I_2 \quad = \quad \{2\}$ oder auch:

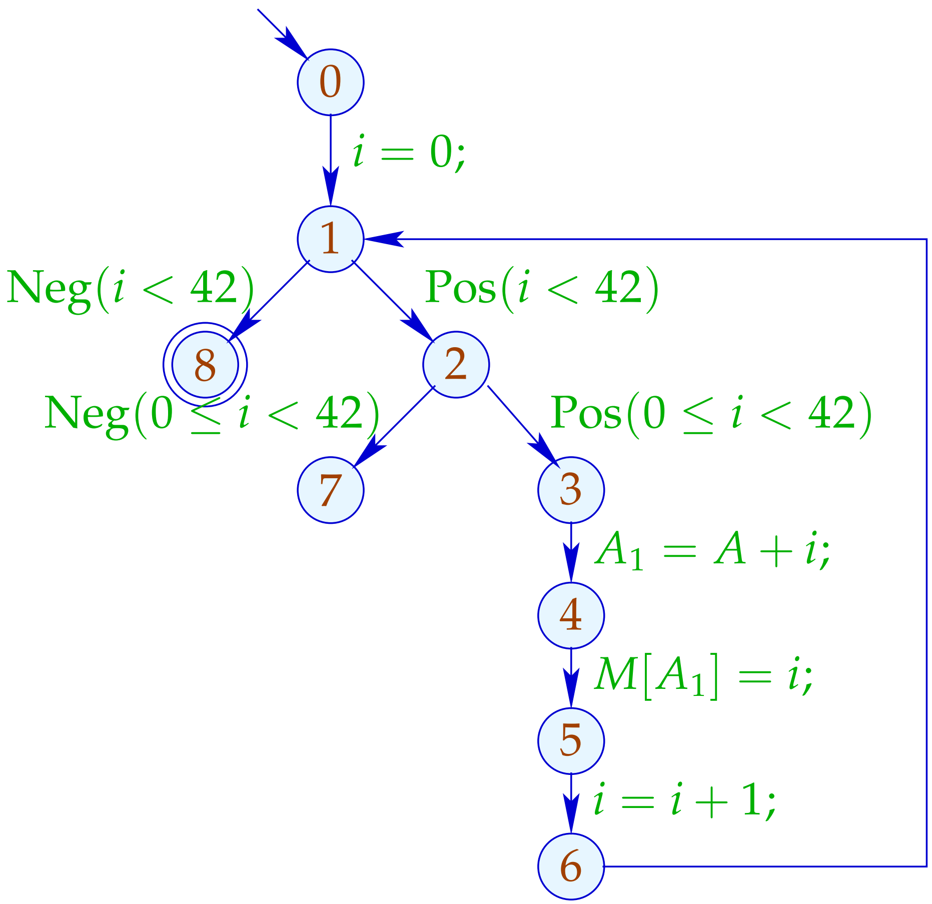
$I_3 \quad = \quad \{3\}$

Die Analyse mit $\quad I = \{1\}$:



	1		2		3	
	l	u	l	u	l	u
0	$-\infty$	$+\infty$	$-\infty$	$+\infty$	dito	
1	0	0	0	$+\infty$		
2	0	0	0	41		
3	0	0	0	41		
4	0	0	0	41		
5	0	0	0	41		
6	1	1	1	42		
7	$\perp$		$\perp$			
8	$\perp$		42	$+\infty$		

Die Analyse mit $I = \{2\}$:



	1		2		3	
	l	u	l	u	l	u
0	$-\infty$	$+\infty$	$-\infty$	$+\infty$	dito	
1	0	0	0	42		
2	0	0	0	$+\infty$		
3	0	0	0	41		
4	0	0	0	41		
5	0	0	0	41		
6	1	1	1	42		
7	$\perp$		42	$+\infty$		
8	$\perp$		42	42		

Diskussion:

- Beide Analysen-Läufe berechnen interessante Informationen :-)
- Der Lauf mit $I = \{2\}$ belegt, dass nach Verlassen der Schleife stets $i = 42$ gilt.
- Nur der Lauf mit $I = \{1\}$ belegt aber, dass der äußere Test den inneren überflüssig macht :-(

Wie findet man einen geeigneten Loop Separator I ???

Idee 3: Narrowing

Sei $\underline{x}$ irgend eine Lösung von (1) , d.h.

$$x_i \sqsupseteq f_i\,\underline{x}\ , \qquad i = 1, \ldots, n$$

Dann gilt für monotone f_i ,

$$\underline{x} \sqsupseteq F\,\underline{x} \sqsupseteq F^2\,\underline{x} \sqsupseteq \ldots \sqsupseteq F^k\,\underline{x} \sqsupseteq \ldots$$

// Narrowing Iteration

Idee 3: Narrowing

Sei $\underline{x}$ irgend eine Lösung von (1) , d.h.

$$x_i \sqsupseteq f_i\,\underline{x} \;, \qquad i = 1, \ldots, n$$

Dann gilt für monotone f_i ,

$$\underline{x} \sqsupseteq F\,\underline{x} \sqsupseteq F^2\,\underline{x} \sqsupseteq \ldots \sqsupseteq F^k\,\underline{x} \sqsupseteq \ldots$$

// Narrowing Iteration

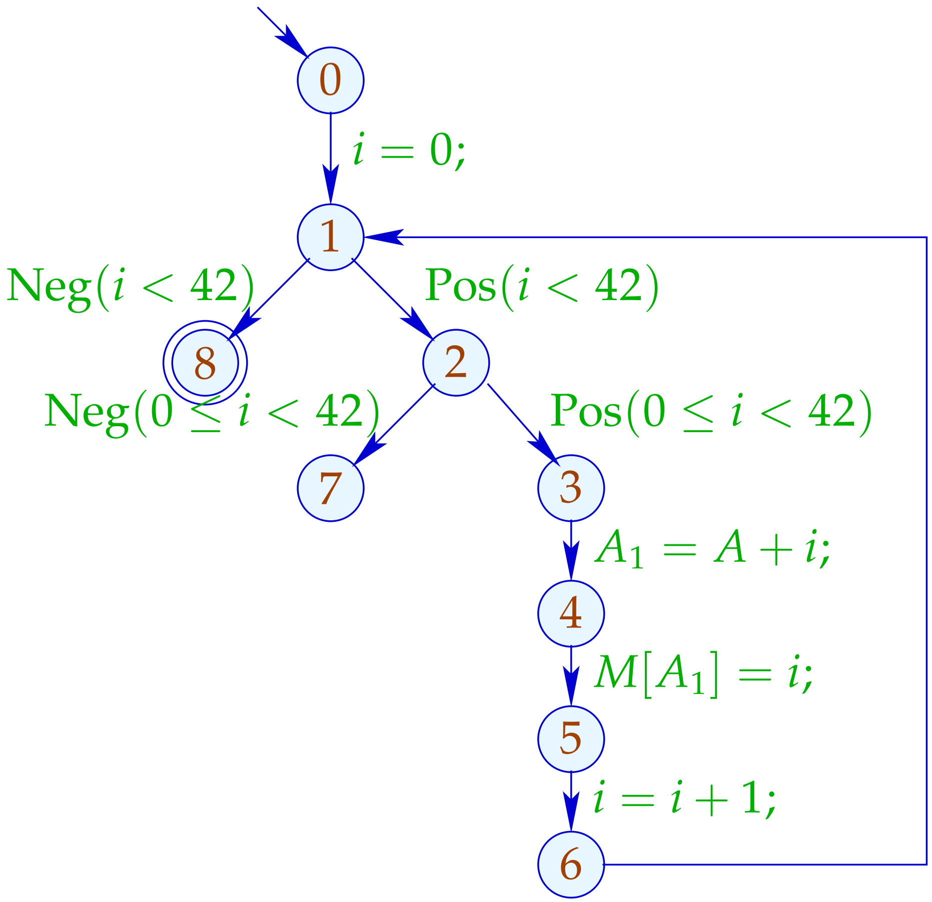
Jeder der Tupel $F^k\,\underline{x}$ ist eine Lösung von (1) :-)

$\Longrightarrow$

Terminierung ist kein Problem mehr:
wir stoppen, wenn wir keine Lust mehr haben :-))

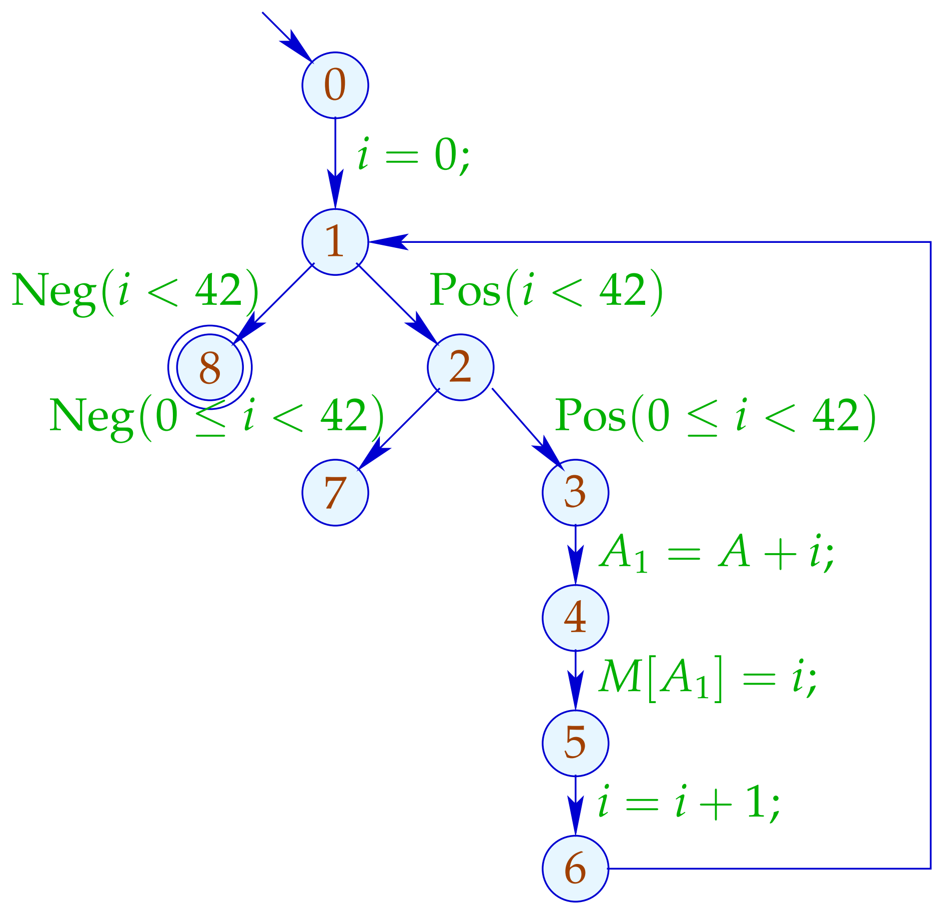
// Analoges gilt für RR-Iteration.

Narrowing Iteration im Beispiel:



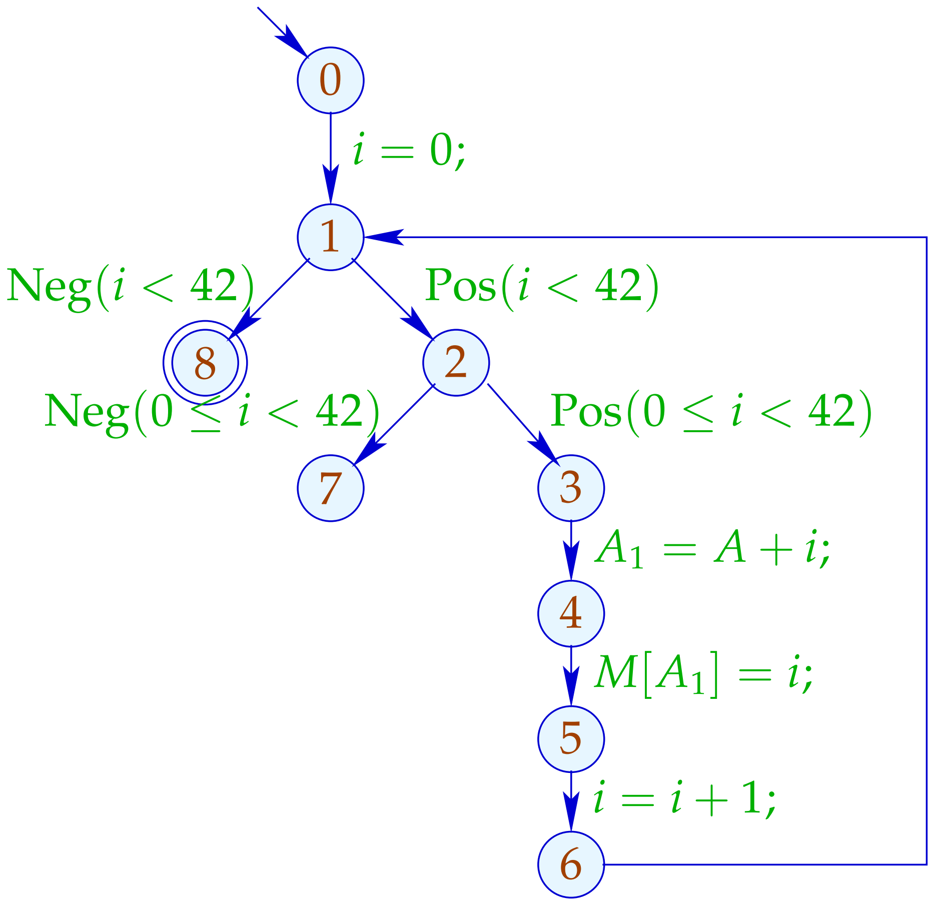
	0	
	l	u
0	$-\infty$	$+\infty$
1	0	$+\infty$
2	0	$+\infty$
3	0	$+\infty$
4	0	$+\infty$
5	0	$+\infty$
6	1	$+\infty$
7	42	$+\infty$
8	42	$+\infty$

Narrowing Iteration im Beispiel:



	0		1	
	l	u	l	u
0	$-\infty$	$+\infty$	$-\infty$	$+\infty$
1	0	$+\infty$	0	$+\infty$
2	0	$+\infty$	0	41
3	0	$+\infty$	0	41
4	0	$+\infty$	0	41
5	0	$+\infty$	0	41
6	1	$+\infty$	1	42
7	42	$+\infty$	$\perp$	
8	42	$+\infty$	42	$+\infty$

Narrowing Iteration im Beispiel:



	0		1		2	
	l	u	l	u	l	u
0	$-\infty$	$+\infty$	$-\infty$	$+\infty$	$-\infty$	$+\infty$
1	0	$+\infty$	0	$+\infty$	0	42
2	0	$+\infty$	0	41	0	41
3	0	$+\infty$	0	41	0	41
4	0	$+\infty$	0	41	0	41
5	0	$+\infty$	0	41	0	41
6	1	$+\infty$	1	42	1	42
7	42	$+\infty$	$\perp$		$\perp$	
8	42	$+\infty$	42	$+\infty$	42	42

Diskussion:

- $\longrightarrow$ Wir beginnen mit einer sicheren Approximation.
- $\longrightarrow$ Wir finden, dass die innere Abfrage redundant ist :-)
- $\longrightarrow$ Wir finden, dass nach der Iteration gilt: $i = 42$:-))
- $\longrightarrow$ Dazu war nicht erforderlich, einen optimalen Loop Separator zu berechnen :-)))

Letzte Frage:

Müssen wir hinnehmen, dass Narrowing möglicherweise nicht terminiert ???

4. Idee: Beschleunigtes Narrowing

Nehmen wir an, wir hätten eine Lösung $\underline{x} = (x_1, \ldots, x_n)$ des Constraint-Systems:

$$x_i \sqsupseteq f_i(x_1, \ldots, x_n), \quad i = 1, \ldots, n \tag{1}$$

Dann schreiben betrachten wir das Gleichungssystem:

$$x_i = x_i \sqcap f_i(x_1, \ldots, x_n), \quad i = 1, \ldots, n \tag{4}$$

Offenbar gilt für monotone f_i : $H^k \underline{x} = F^k \underline{x}$:-)

wobei $H(x_1, \ldots, x_n) = (y_1, \ldots, y_n), \quad y_i = x_i \sqcap f_i(x_1, \ldots, x_n)$.

In (4) ersetzen wir $\sqcap$ durch den neuen Operator $\overline{\sqcap}$ mit:

$$a_1 \sqcap a_2 \sqsubseteq a_1 \mathbin{\overline{\sqcap}} a_2 \sqsubseteq a_1$$

... für die Intervall-Analyse:

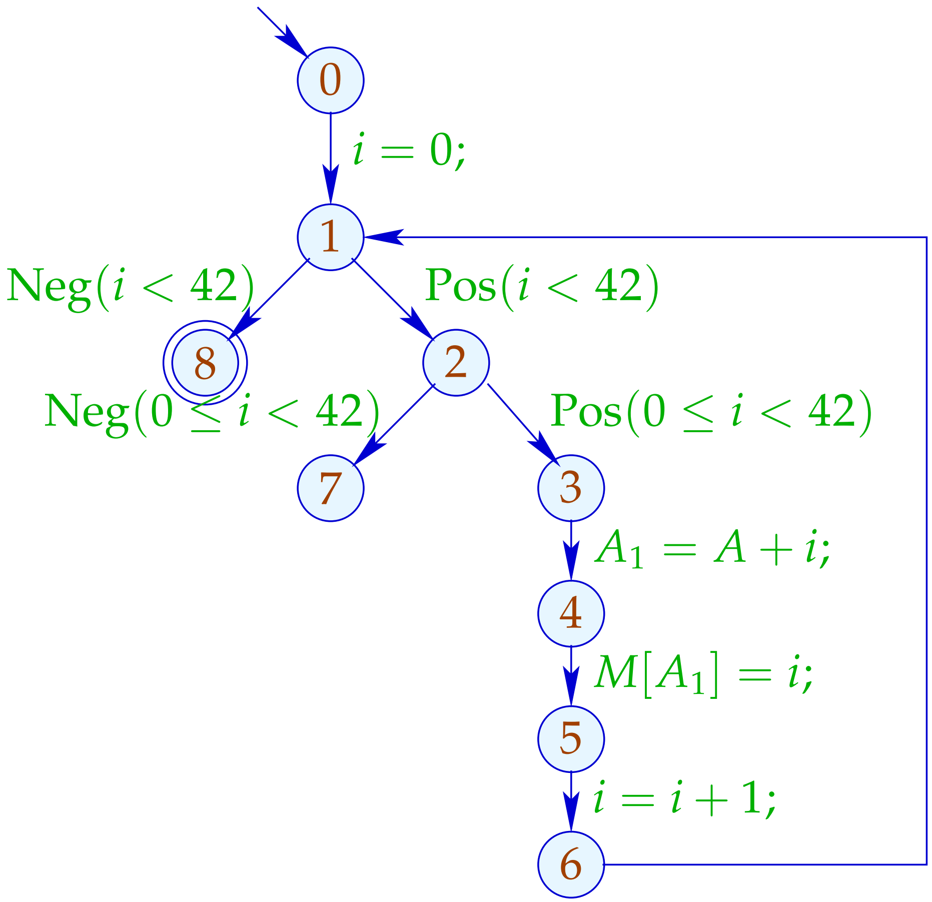
Wir konservieren endliche Intervall-Grenzen :-)

Deshalb $\quad \bot \sqcap D = D \sqcap \bot = \bot \quad$ und für $\quad D_1 \neq \bot \neq D_2$:

$$
\begin{array}{rcl}
(D_1 \sqcap D_2)\, x & = & (D_1\, x) \sqcap (D_2\, x) \qquad \text{wobei} \\
{[l_1, u_1]} \sqcap [l_2, u_2] & = & [l, u] \qquad \text{mit} \\
l & = & \begin{cases} l_2 & \text{falls} \quad l_1 = -\infty \\ l_1 & \text{sonst} \end{cases} \\
u & = & \begin{cases} u_2 & \text{falls} \quad u_1 = \infty \\ u_1 & \text{sonst} \end{cases}
\end{array}
$$

$\Longrightarrow \quad \sqcap \quad$ ist nicht kommutativ !!!

Beschleunigtes Narrowing im Beispiel:



	0		1		2	
	l	u	l	u	l	u
0	$-\infty$	$+\infty$	$-\infty$	$+\infty$	$-\infty$	$+\infty$
1	0	$+\infty$	0	$+\infty$	0	42
2	0	$+\infty$	0	41	0	41
3	0	$+\infty$	0	41	0	41
4	0	$+\infty$	0	41	0	41
5	0	$+\infty$	0	41	0	41
6	1	$+\infty$	1	42	1	42
7	42	$+\infty$	$\perp$		$\perp$	
8	42	$+\infty$	42	$+\infty$	42	42

Diskussion:

→ Achtung: Widening liefert für nicht-monotone f_i eine Lösung. Narrowing liefert dagegen nur für monotone f_i eine Lösung!!

→ Das beschleunigte Narrowing liefert (im Beispiel) das richtige Ergebnis :-)

→ Erlaubt der neue Operator $\sqcap\!\!\!\!\!\!-$ nur endlich viele Verbesserungen bei jedem Wert, kann Narrowing bis zur Stabilisierung durchgeführt werden.

→ Für die Intervall-Analyse sind das maximal

$$\#\mathit{Punkte} \cdot (1 + 2 \cdot \#\mathit{Vars})$$

1.6 Pointer-Analyse

Fragen:

→ Sind zwei Adressen möglicherweise gleich?

→ Sind zwei Adressen definitiv gleich?

1.6 Pointer-Analyse

Fragen:

- $\rightarrow$ Sind zwei Adressen möglicherweise gleich? May Alias
- $\rightarrow$ Sind zwei Adressen definitiv gleich? Must Alias

$\Longrightarrow$ Alias-Analyse

Die bisherigen Analysen ohne Alias-Information:

(1) Verfügbare Ausdrücke:

- Erweitere die Menge $Expr$ der Ausdrücke um die vorkommenden Loads $M[R]$.

- Erweitere die Kanten-Effekte:

$$
\begin{array}{lll}
[\![x = e;]\!]^\sharp A & = & (A \cup \{e\}) \backslash Expr_x \\
[\![x = M[R];]\!]^\sharp A & = & (A \cup \{M[R]\}) \backslash Expr_x \\
[\![M[R] = x;]\!]^\sharp A & = & A \backslash Loads
\end{array}
$$

(2) Werte von Variablen:

- Erweitere die Menge *Expr* der Ausdrücke um die vorkommenden Loads $M[R]$.

- Erweitere die Kanten-Effekte:

$$
\begin{array}{lll}
[\![x = M[R];]\!]^\sharp\, V\, e & = & \begin{cases} \{x\} & \text{falls} \quad e = M[R] \\ V\, e \backslash \{x\} & \text{sonst} \end{cases} \\
[\![M[R] = x;]\!]^\sharp\, V & = & V
\end{array}
$$

(3) Konstantenpropagation:

- Erweitere den abstrakten Zustand um einen abstrakten Speicher M

- Führe Speicher-Operationen mit bekannten Adressen aus!

$$[\![x = M[R];]\!]^\sharp\,(D, M) = \begin{cases} (D \oplus \{x \mapsto M\,a\}, M) & \text{falls} \quad D\,R = a \sqsubset \top \\ (D \oplus \{x \mapsto \top\}, M) & \text{sonst} \end{cases}$$

$$[\![M[R] = x;]\!]^\sharp\,(D, M) = \begin{cases} (D, M \oplus \{a \mapsto D\,x\}) & \text{falls} \quad D\,R = a \sqsubset \top \\ (D, \underline{\top}) & \text{sonst} \end{cases} \qquad \text{wobei}$$

$$\underline{\top}\,a = \top \qquad (a \in \mathbb{N})$$

Probleme:

- Adressen sind aus $\mathbb{N}$:-(

  Es gibt zwar keine unendliche aufsteigende Ketten, aber ...
- Exakte Adressen sind zur Compilezeit selten bekannt :-(
- Am selben Programmpunkt wird i.a. auf mehrere Adressen zugegriffen ...
- Abspeichern an unbekannter Adresse zerstört alle Information M :-(

$\Longrightarrow$ Konstanten-Propagation versagt :-(

$\Longrightarrow$ Speicherzugriffe/Pointer zerstören Präzision :-(

Vereinfachung:

- Wir betrachten Pointer auf Strukturen mit Komponenten a, b :-)
- Wir verzichten auf Wohl-Getyptheit dieser Komponenten.
- Neue Statements:

$$x = \mathsf{new}(); \quad // \quad \text{Allokation eines neuen Paars}$$
$$x = R \to a; \quad // \quad \text{Laden einer Komponente}$$
$$R \to a = x; \quad // \quad \text{Setzen einer Komponente}$$

- Wir verzichten auf Pointer-Arithmetik :-)

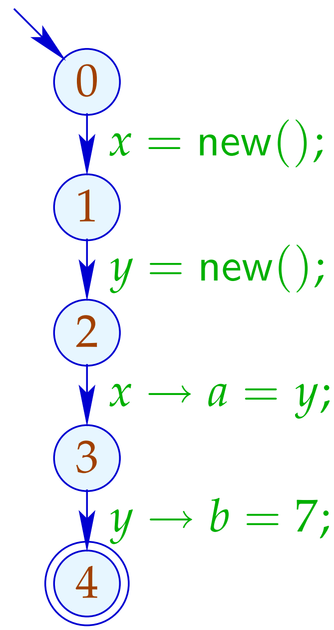
Einfaches Beispiel:

$x = \mathsf{new}();$

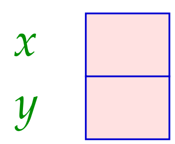
$y = \mathsf{new}();$

$x \to a = y;$

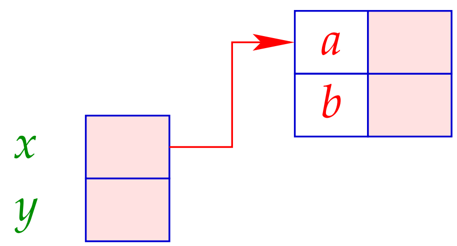
$y \to b = 7;$



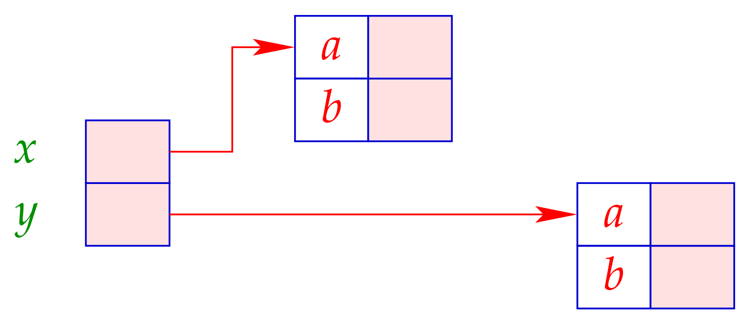
Die Semantik:



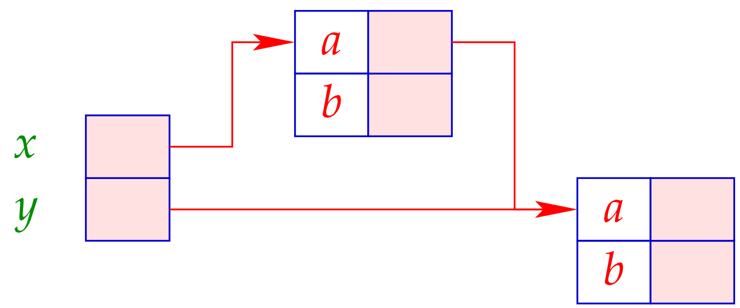
Die Semantik:



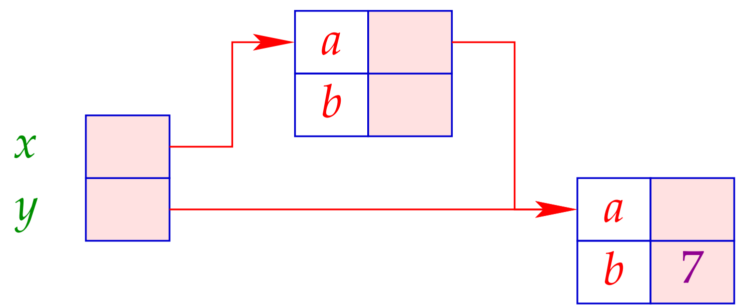
Die Semantik:



Die Semantik:



Die Semantik:



Schwierigeres Beispiel:

$r = \mathsf{Null};$

$\mathsf{while}\ (t \neq \mathsf{Null})\ \{$

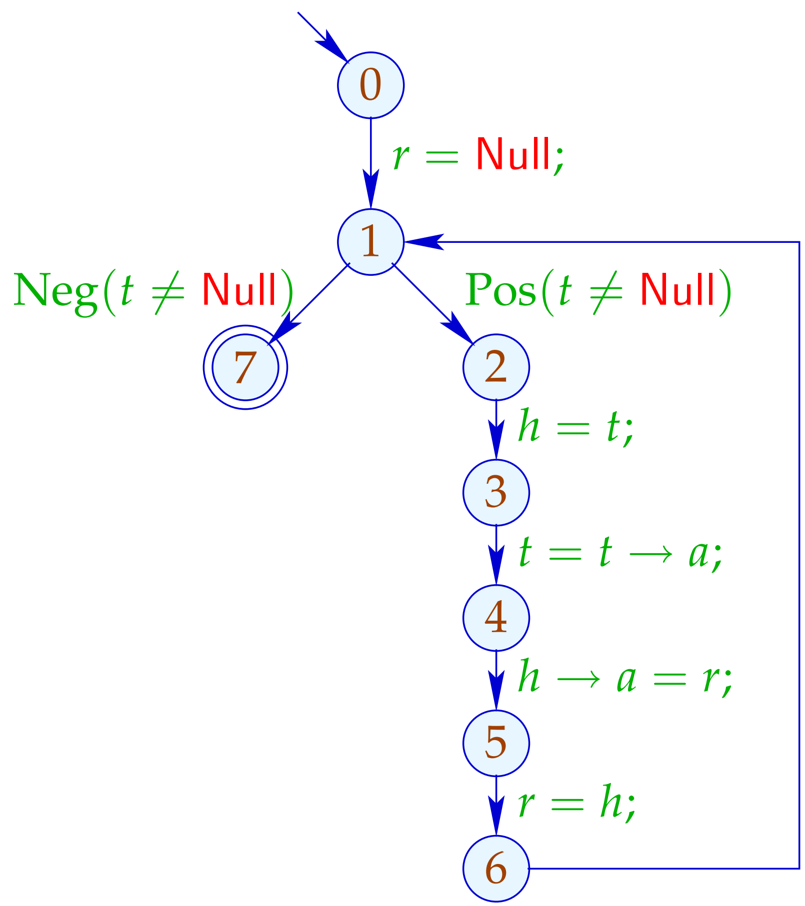
$\quad h = t;$

$\quad t = t \to a;$

$\quad h \to a = r;$

$\quad r = h;$

$\}$



Konkrete Semantik:

Ein Speicher ist jetzt eine endliche Ansammlung von Paaren.

Nach h new-Operationen haben wir:

$$\begin{array}{lcll} Addr_h & = & \{\mathsf{ref}\; a \mid 0 \leq a < h\} & // \quad \text{Adressen} \\ Val_h & = & Addr_h \cup \mathbb{Z} & // \quad \text{Werte} \\ Store_h & = & (Addr_h \times \{a, b\}) \to Val_h & // \quad \text{Speicher} \\ State_h & = & (Vars \to Val_h) \times Store_h & // \quad \text{Zustände} \end{array}$$

Der Einfachheit setzen wir: $\quad 0 = \mathsf{Null}$

Sei $(\rho, \mu) \in State_h$. Dann erhalten wir für die neuen Kanten:

$$
\begin{array}{lll}
[\![x = \mathsf{new}();]\!]\, (\rho, \mu) & = & (\rho \oplus \{x \mapsto \mathsf{ref}\, h\}, \\
 & & \quad \mu \oplus \{(\mathsf{ref}\, h).a \mapsto 0, (\mathsf{ref}\, h).b \mapsto 0\}) \\
[\![x = R \to a;]\!]\, (\rho, \mu) & = & (\rho \oplus \{x \mapsto \mu\,((\rho\, R).a)\}, \mu) \\
[\![R \to a = x;]\!]\, (\rho, \mu) & = & (\rho, \mu \oplus \{(\rho\, R).a \mapsto \rho\, x\})
\end{array}
$$

Achtung:

Diese Semantik ist zu detailliert, weil sie mit absoluten Adressen rechnet. Die beiden Programme:

$$x = \mathsf{new}(); \qquad\qquad y = \mathsf{new}();$$
$$y = \mathsf{new}(); \qquad\qquad x = \mathsf{new}();$$

werden nicht als äquivalent betrachtet !!?

Ausweg:

Definiere Äquivalenz bis auf Permutation von Adressen :-)

Alias-Analyse 1. Idee:

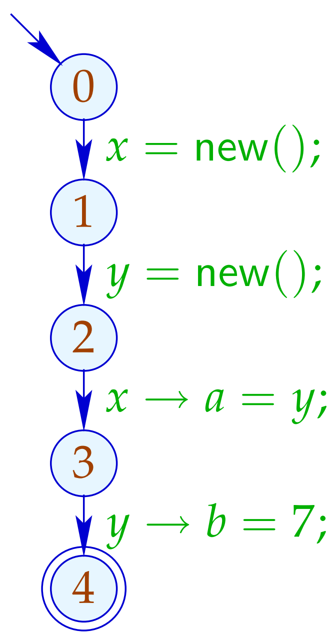
- Unterscheide endlich viele verschiedene Klassen von Objekten im Speicher.
- Benutze Mengen von Adressen als abstrakte Werte!

$\Longrightarrow$ Points-to-Analyse

$$
\begin{array}{lcll}
\mathit{Addr}^\sharp & = & \mathit{Edges} & // \text{ Erzeugungs-Kanten} \\
\mathit{Val}^\sharp & = & 2^{\mathit{Addr}^\sharp} & // \text{ Abstrakte Werte} \\
\mathit{Store}^\sharp & = & (\mathit{Addr}^\sharp \times \{a, b\}) \to \mathit{Val}^\sharp & // \text{ abstrakter Speicher} \\
\mathit{State}^\sharp & = & (\mathit{Vars} \to \mathit{Val}^\sharp) \times \mathit{Store}^\sharp & // \text{ Zustände}
\end{array}
$$

// vollständiger Verband !!!

... im einfachen Beispiel:



	x	y	$(0,1).a$
0	$\emptyset$	$\emptyset$	$\emptyset$
1	$\{(0,1)\}$	$\emptyset$	$\emptyset$
2	$\{(0,1)\}$	$\{(1,2)\}$	$\emptyset$
3	$\{(0,1)\}$	$\{(1,2)\}$	$\{(1,2)\}$
4	$\{(0,1)\}$	$\{(1,2)\}$	$\{(1,2)\}$

Die Kanten-Effekte:

$$
\begin{array}{lcl}
[\![(\_, ;, \_)]\!]^\sharp\,(D, M) & = & (D, M) \\
[\![(\_, \mathrm{Pos}(e), \_)]\!]^\sharp\,(D, M) & = & (D, M) \\
[\![(\_, x = y;, \_)]\!]^\sharp\,(D, M) & = & (D \oplus \{x \mapsto D\,y\}, M) \\
[\![(\_, x = e;, \_)]\!]^\sharp\,(D, M) & = & (D \oplus \{x \mapsto \emptyset\}, M) \qquad , \qquad e \notin \mathit{Vars} \\
\\
[\![(u, x = \mathsf{new}();, v)]\!]^\sharp\,(D, M) & = & (D \oplus \{x \mapsto \{(u, v)\}\}, M) \\
[\![(\_, x = R \to a;, \_)]\!]^\sharp\,(D, M) & = & (D \oplus \{x \mapsto \bigcup\{M(f.a) \mid f \in D\,R\}\}, M) \\
[\![(\_, R \to a = x;, \_)]\!]^\sharp\,(D, M) & = & (D, M \oplus \{f.a \mapsto (M\,(f.a) \cup D\,x) \mid f \in D\,R\}
\end{array}
$$

Achtung:

- Den Wert Null haben wir nicht mit-modelliert. Dereferenzieren von Null kann darum nicht entdeckt werden :-(
- Destruktive Updates sind nur von Variablen möglich, nicht im Speicher!

  $\Longrightarrow$ keine Information, falls Speicher-Objekte nicht vorinitialisiert sind :-((
- Die Kanten-Effekte hängen jetzt von der ganzen Kante ab.

  Die Analyse lässt sich so nicht gegenüber der Referenz-Semantik als korrekt erweisen :-(

  Zur Korrektheit muss die konkrete Semantik mit zusätzlicher Information instrumentiert werden, die vermerkt, an welchem Programmpunkt eine Adresse erzeugt wurde.

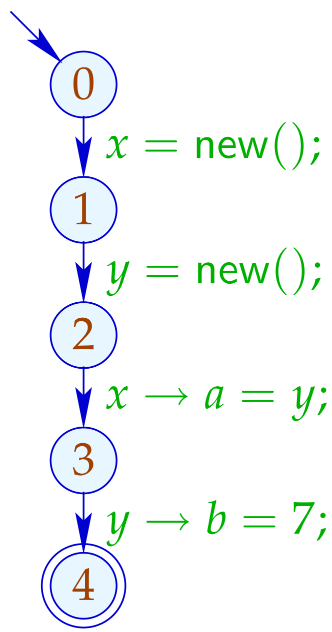
...

- Wir berechnen mögliche Points-to-Information.
- Daraus können wir May-Alias-Information gewinnen.
- Die Analyse kann jedoch ziemlich aufwendig sein (ohne viel raus zu kriegen :-(
- Separate Information für jeden Programmpunkt ist möglicherweise nicht nötig ??

Alias-Analyse 2. Idee:

Berechne für jede Variable und jede Adresse einen Wert, der die Werte an sämtlichen Programmpunkten sicher approximiert!

... im einfachen Beispiel:



x	$\{(0,1)\}$
y	$\{(1,2)\}$
$(0,1).a$	$\{(1,2)\}$
$(0,1).b$	$\emptyset$

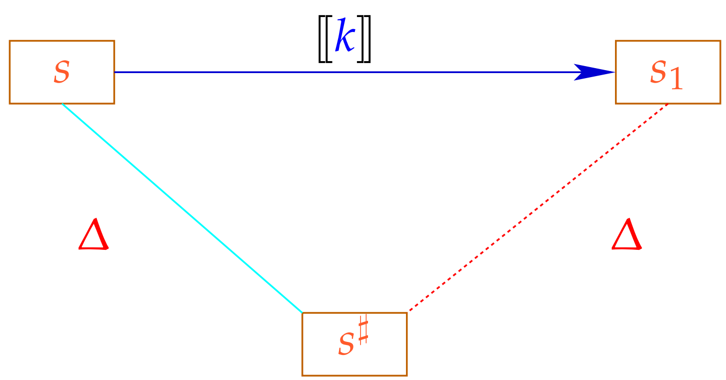
Jede Kante (u, lab, v) gibt Anlass zu Constraints:

lab	*Constraints*
$x = y;$	$\mathcal{P}[x] \supseteq \mathcal{P}[y]$
$x = \mathsf{new}();$	$\mathcal{P}[x] \supseteq \{(u, v)\}$
$x = R \to a;$	$\mathcal{P}[x] \supseteq \bigcup\{\mathcal{P}[f.a] \mid f \in \mathcal{P}[R]\}$
$R \to a = x;$	$\mathcal{P}[f.a] \supseteq (f \in \mathcal{P}[R])\,?\,\mathcal{P}[x] : \emptyset$ für alle $f \in \mathit{Addr}^\sharp$

Andere Kanten haben keinen Effekt :-)

Diskussion:

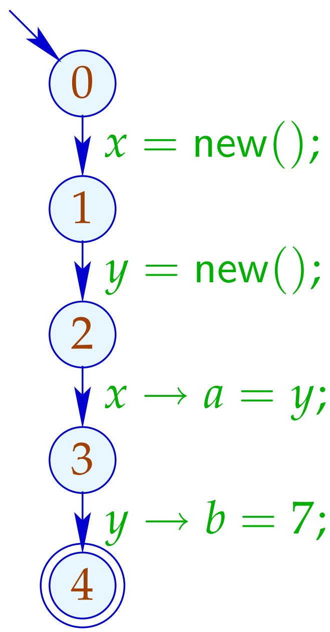
- Das resultierende Constraint-System ist $\mathcal{O}(k \cdot n)$ bei k abstrakten Adressen und n Kanten :-(
- Die Anzahl eventuell notwendiger Iterationen ist $\mathcal{O}(k)$ ...
- Die berechnete Information ist möglicherweise immer noch zu präzise !!?
- Zur Korrektheit einer Lösung $s^\sharp \in \mathit{States}^\sharp$ des Constraint-Systems zeigt man:



Alias-Analyse 3. Idee:

Berechne eine Äquivalenzrelation $\equiv$ auf Variablen x und Selektoren $y \to a$ mit $s_1 \equiv s_2$ falls an irgendeinem u s_1, s_2 die gleiche Adresse enthalten ...

... im einfachen Beispiel:



$$\equiv \;=\; \{\{x\},\; \{y, x \to a\},\; \{x \to b\},\; \{y \to b\}\}$$

Diskussion:

→ Wir berechnen eine Information für das ganze Programm.

→ Die Berechnung dieser Information verwaltet Partitionen $\pi = \{P_1, \ldots, P_m\}$:-)

→ Einzelne Mengen P_i identifizieren wir durch einen Repräsentanten $p_i \in P_i$.

→ Die Operationen auf einer Partition π sind:

$$
\begin{array}{lll}
\textsf{find}\,(\pi, p) & = & p_i \qquad\qquad \text{falls } p \in P_i \\
 & // & \text{liefert den Repräsentanten} \\
\textsf{union}\,(\pi, p_{i_1}, p_{i_2}) & = & \{P_{i_1} \cup P_{i_2}\} \cup \{P_j \mid i_1 \neq j \neq i_2\} \\
 & // & \text{vereinigt repräsentierte Klassen}
\end{array}
$$

$\longrightarrow$ Sind $x_1, x_2 \in Vars$ äquivalent, müssen auch $x_i \to a$ und $x_i \to b$ äquivalent sein :-)

$\longrightarrow$ Ist $P_i \cap Vars \neq \emptyset$, soll auch $p_i \in Vars$ gelten. Dann können wir union rekursiv anwenden :

$$
\begin{array}{lll}
\mathsf{union}^*\,(\pi, q_1, q_2) & = & \mathsf{let}\ \ p_{i_1} \ = \ \mathsf{find}\,(\pi, q_1) \\
& & \phantom{\mathsf{let}\ \ } p_{i_2} \ = \ \mathsf{find}\,(\pi, q_2) \\
& & \mathsf{in}\ \ \mathsf{if}\ p_{i_1} == p_{i_2}\ \mathsf{then}\ \pi \\
& & \phantom{\mathsf{in}\ \ } \mathsf{else}\ \ \mathsf{let}\ \pi\ = \ \mathsf{union}\,(\pi, p_{i_1}, p_{i_2}) \\
& & \phantom{\mathsf{in}\ \ \mathsf{else}\ \ } \mathsf{in}\ \ \mathsf{if}\ p_{i_1}, p_{i_2} \in Vars\ \mathsf{then} \\
& & \phantom{\mathsf{in}\ \ \mathsf{else}\ \ \mathsf{in}\ \ } \mathsf{let}\ \pi = \mathsf{union}^*\,(\pi, p_{i_1} \to a, p_{i_2} \to a) \\
& & \phantom{\mathsf{in}\ \ \mathsf{else}\ \ \mathsf{in}\ \ } \mathsf{in}\ \mathsf{union}^*\,(\pi, p_{i_1} \to b, p_{i_2} \to b) \\
& & \phantom{\mathsf{in}\ \ \mathsf{else}\ \ \mathsf{in}\ \ } \mathsf{else}\ \pi
\end{array}
$$

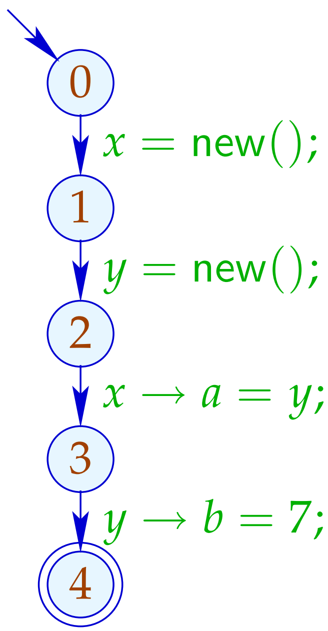
Die Analyse iteriert einmal über alle Kanten:

$$\pi = \{\{x\}, \{x \to a\}, \{x \to b\} \mid x \in Vars\};$$
$$\text{forall } k = (\_, lab, \_) \text{ do } \quad \pi = [\![lab]\!]^\sharp\, \pi;$$

Dabei ist:

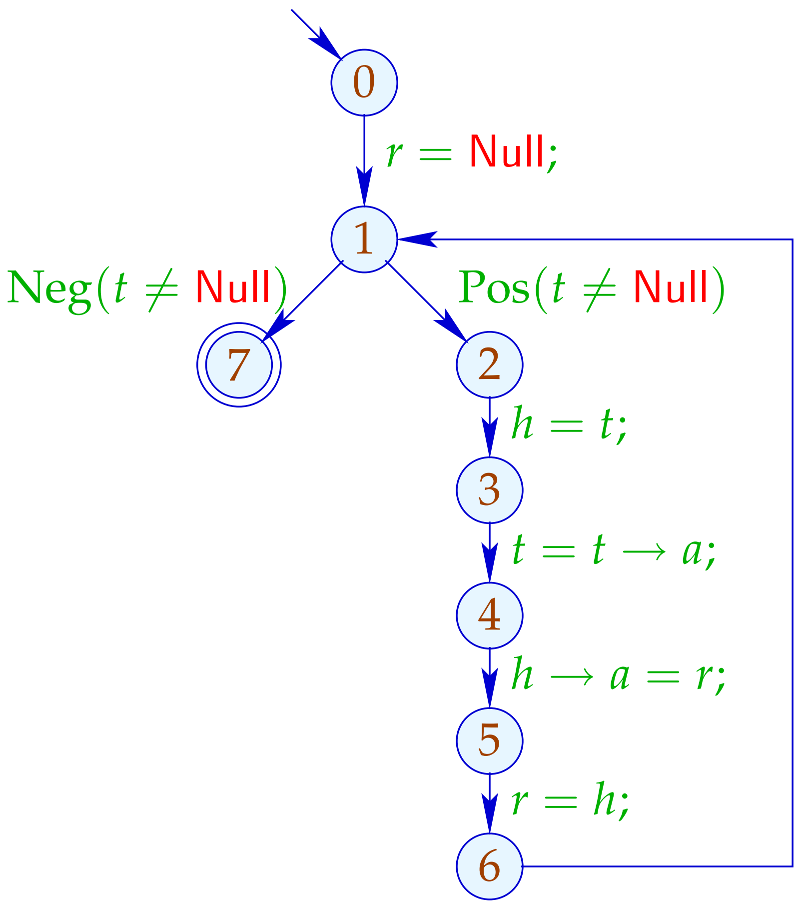
$$\begin{array}{lcll}
[\![x = y;]\!]^\sharp\, \pi & = & \text{union}^*\,(\pi, x, y) & \\
[\![x = R \to a;]\!]^\sharp\, \pi & = & \text{union}^*\,(\pi, x, R \to a) & \\
[\![R \to a = x;]\!]^\sharp\, \pi & = & \text{union}^*\,(\pi, x, R \to a) & \\
[\![lab]\!]^\sharp\, \pi & = & \pi & \text{sonst}
\end{array}$$

... im einfachen Beispiel:



	$\{\{x\}, \{y\}, \{x \rightarrow a\}, \{y \rightarrow a\}, \ldots\}$
$(0, 1)$	$\{\{x\}, \{y\}, \{x \rightarrow a\}, \{y \rightarrow a\}, \ldots\}$
$(1, 2)$	$\{\{x\}, \{y\}, \{x \rightarrow a\}, \{y \rightarrow a\}, \ldots\}$
$(2, 3)$	$\{\{x\}, \boxed{\{y, x \rightarrow a\}}, \{y \rightarrow a\}, \ldots\}$
$(3, 4)$	$\{\{x\}, \{y, x \rightarrow a\}, \{y \rightarrow a\}, \ldots\}$

... im komplizierten Beispiel:



	$\{\{h\},\{r\},\{t\},\{h \to a\},\{t \to a\}\}$
$(2,3)$	$\{\boxed{\{h,t\}},\{r\},\boxed{\{h \to a, t \to a\}}\}$
$(3,4)$	$\{\boxed{\{h,t,h \to a, t \to a\}},\{r\}\}$
$(4,5)$	$\{\boxed{\{h,t,r,h \to a, t \to a\}}\}$
$(5,6)$	$\{\{h,t,r,h \to a, t \to a\}\}$

Achtung:

Um überhaupt etwas heraus zu kriegen, müssen wir annehmen, dass alle Variablen anfangs auf verschiedene Adressen zeigen.

Zur Komplexität:

Wir haben:

$\mathcal{O}(\#\, \mathit{Kanten})$	Aufrufe von	union*
$\mathcal{O}(\#\, \mathit{Kanten})$	Aufrufe von	find
$\mathcal{O}(\#\, \mathit{Vars})$	Aufrufe von	union

$\Longrightarrow$ Wir benötigen effiziente Union-Find-Datenstruktur :-)

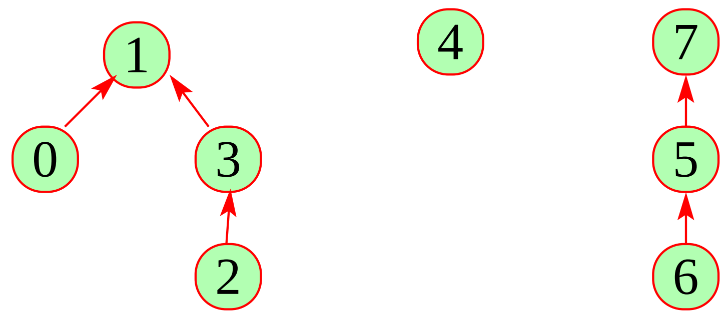
Idee:

Repräsentiere Partition von U als gerichteten Wald:

- Zu $u \in U$ verwalten wir einen Vater-Verweis $F[u]$.
- Elemente u mit $F[u] = u$ sind Wurzeln.

Einzelne Bäume sind Äquivalenzklassen.

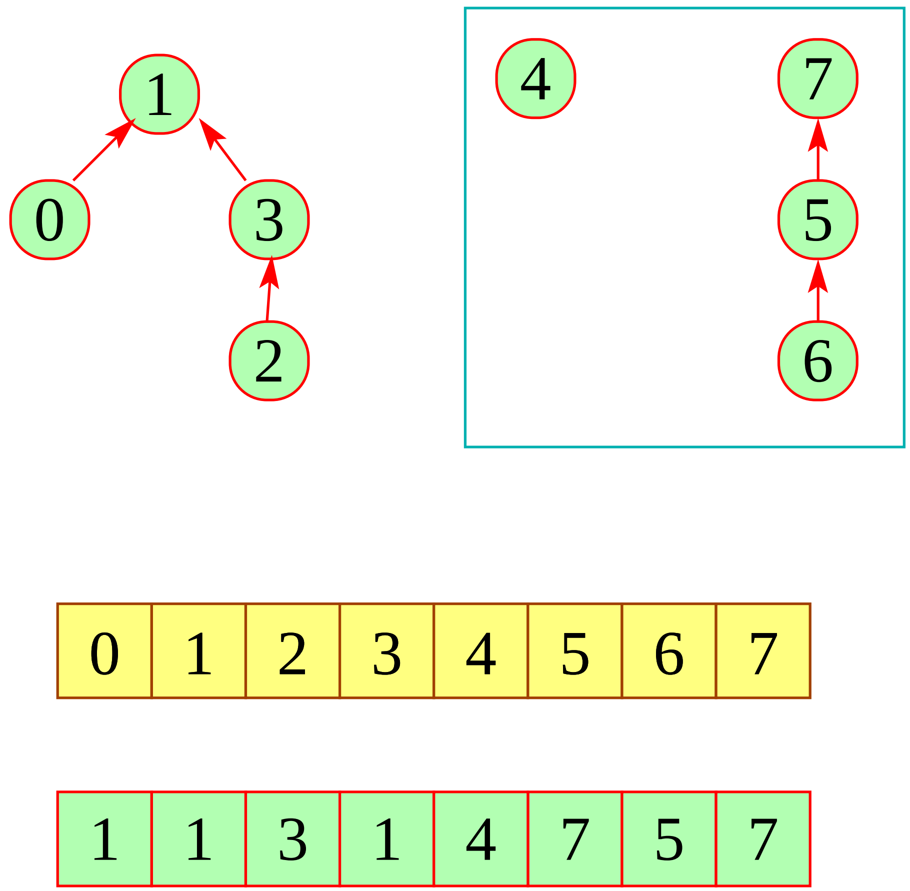
Ihre Wurzeln sind die Repräsentanten ...



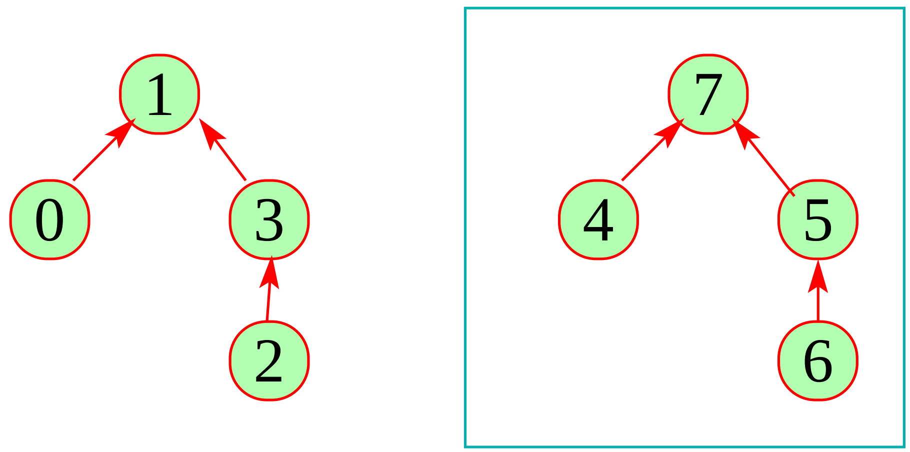
0	1	2	3	4	5	6	7
1	1	3	1	4	7	5	7

$\longrightarrow$ find (π, u) folgt den Vater-Verweisen :-)

$\longrightarrow$ union (π, u_1, u_2) hängt den Vater-Verweis eines u_i um ...



1
0
3
2
4
7
5
6
0 1 2 3 4 5 6 7
1 1 3 1 4 7 5 7



0	1	2	3	4	5	6	7

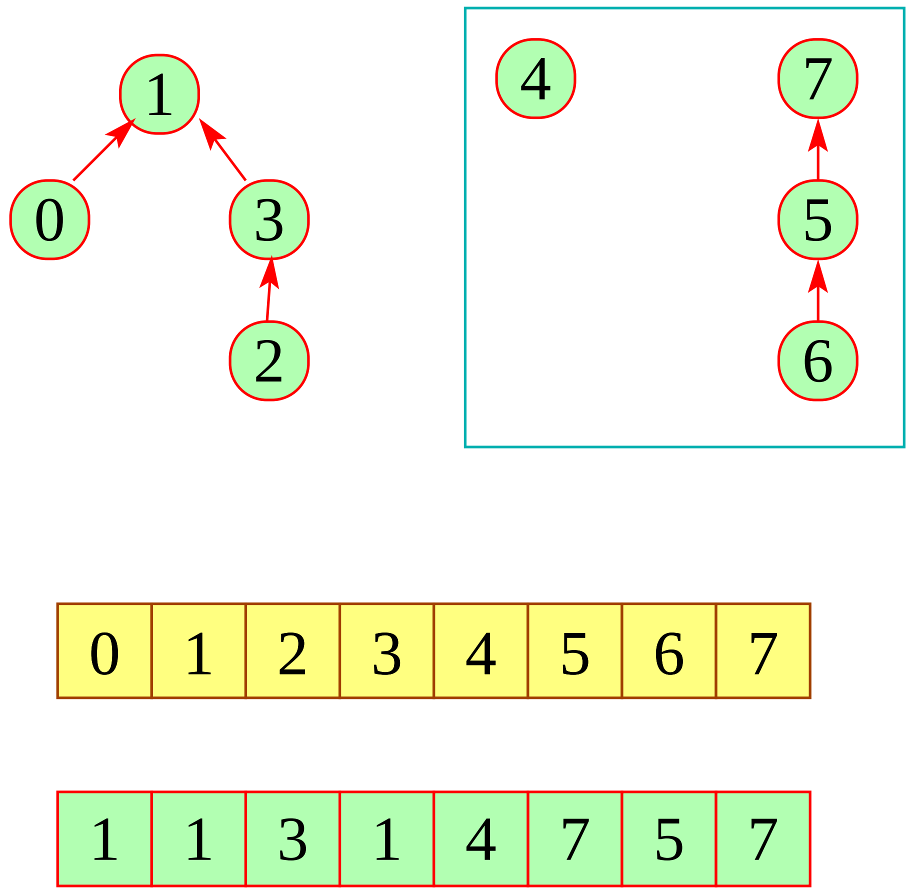
1	1	3	1	7	7	5	7

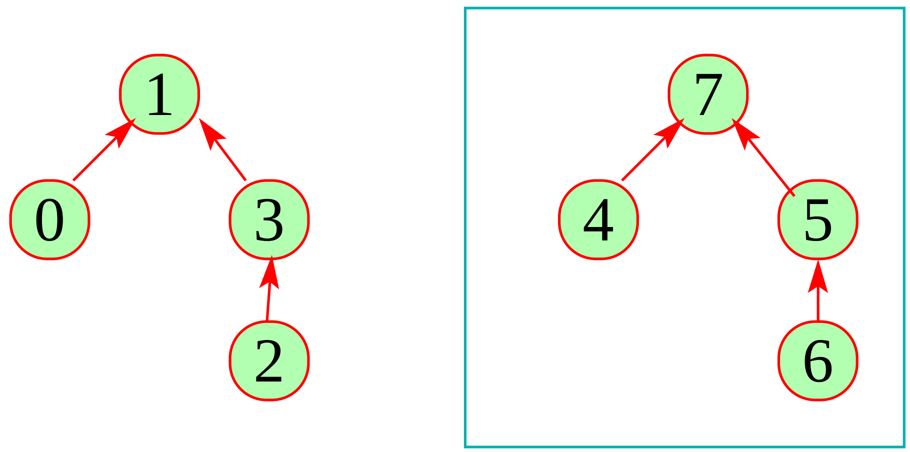
Die Kosten:

union	:	$\mathcal{O}(1)$	:-)
find	:	$\mathcal{O}(depth(\pi))$	:-(

Strategie zur Vermeidung tiefer Bäume:

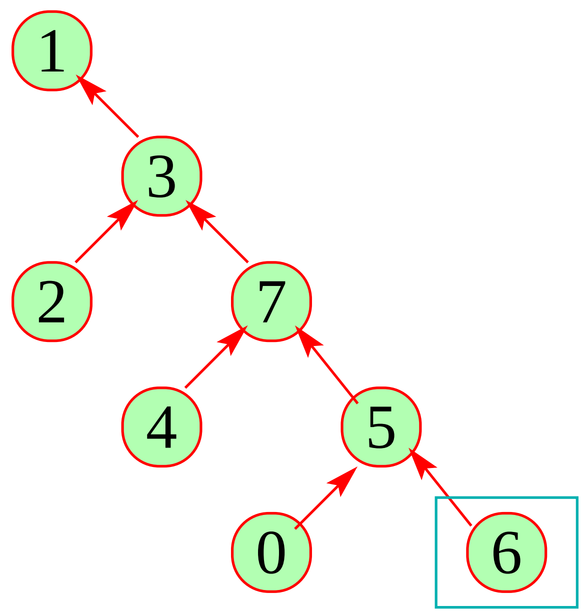
- Hänge den kleineren Baum unter den größeren !
- Benutze find , um Pfade zu komprimieren ...


1
0
3
2
4
7
5
6
0 1 2 3 4 5 6 7
1 1 3 1 4 7 5 7

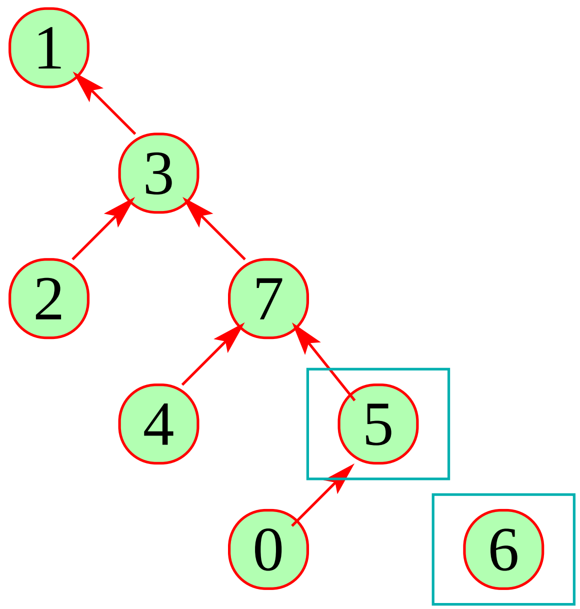


0	1	2	3	4	5	6	7

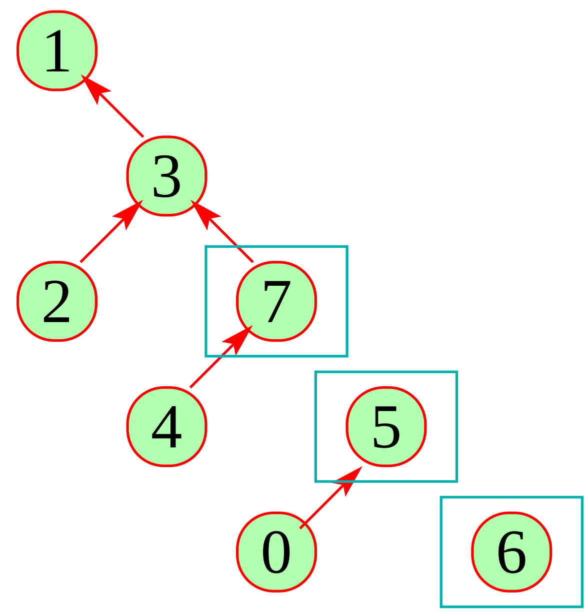
1	1	3	1	7	7	5	7



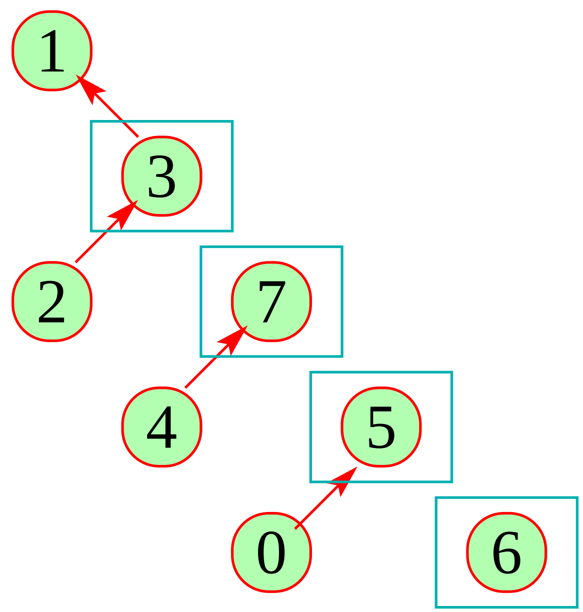
0	1	2	3	4	5	6	7
5	1	3	1	7	7	5	3



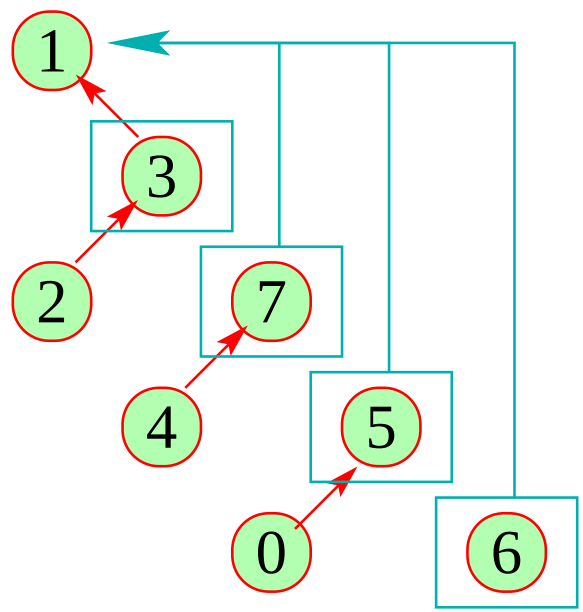
0	1	2	3	4	5	6	7
5	1	3	1	7	7	5	3



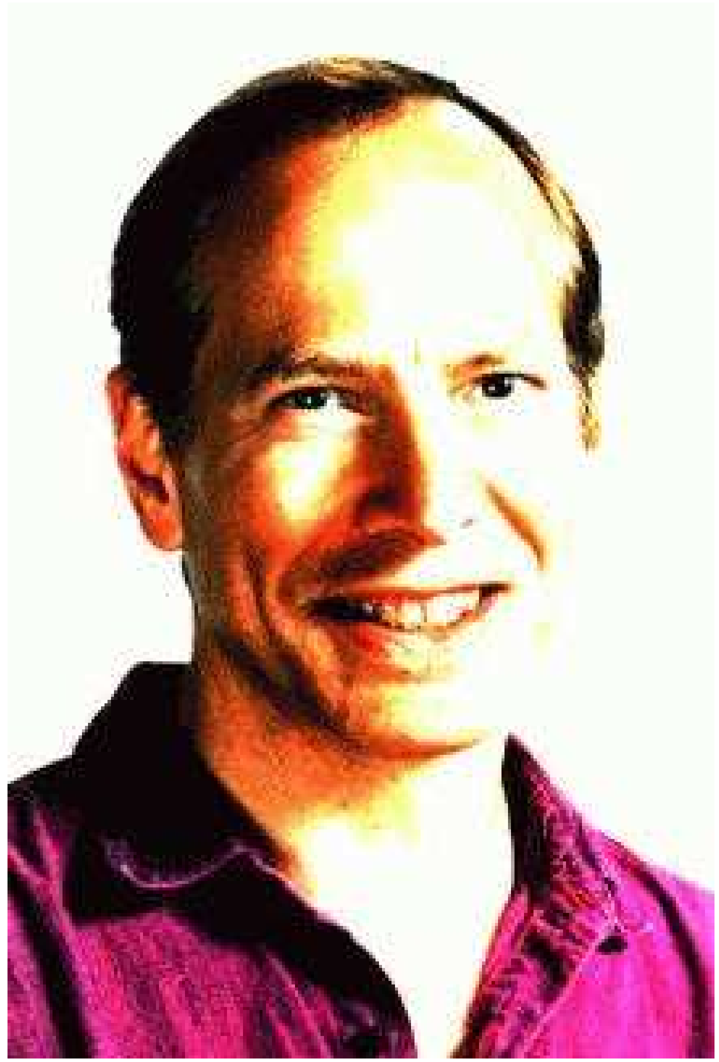
0	1	2	3	4	5	6	7
5	1	3	1	7	7	5	3



0	1	2	3	4	5	6	7
5	1	3	1	7	7	5	3



0	1	2	3	4	5	6	7
5	1	3	1	1	7	1	1



Robert Endre Tarjan, Princeton

Beachte:

- Mit dieser Datenstruktur dauern n union- und m find-Operationen $\mathcal{O}(n + m \cdot \alpha(n, n))$

  // α die inverse Ackermann-Funktion :-)

- Für unsere Anwendung müssen wir union nur so modifizieren, dass an den Wurzeln nach Möglichkeit Elemente aus *Vars* stehen.

- Diese Modifikation vergrößert die asymptotische Laufzeit nicht :-)

Fazit:

Die Analyse ist blitzschnell — findet aber nicht sehr viel heraus.

Exkurs 3: Fixpunkt-Algorithmen

Betrachte: $x_i \sqsupseteq f_i(x_1, \ldots, x_n)\,, \quad i = 1, \ldots, n$

Beobachtung:

RR-Iteration ist ineffizient:

→ Wir benötigen eine ganze Runde, um Terminierung festzustellen :-(

→ Ändert sich in einer Runde der Wert nur einer Variable, berechnen wir trotzdem alle neu :-(

→ Die praktische Laufzeit hängt von der Reihenfolge der Variablen ab :-(

Idee: Workset-Iteration

Ändert eine Variable x_i ihren Wert, werten wir alle Variablen neu aus, die von x_i abhängen. Technisch benötigen wir:

→ die Mengen $Dep\ f_i$ der Variablen, auf die die Auswertung von f_i zugreift. Daraus berechnen wir:

$$I[x_i] = \{x_j \mid x_i \in Dep\ f_j\}$$

d.h. die Menge der x_j, die von x_i abhängen.

→ die Werte $D[x_i]$ der x_i, wobei anfangs $D[x_i] = \perp$;

→ Eine Menge W der Variablen, deren Wert neu berechnet werden muss ...

Der Algorithmus:

$$
\begin{array}{l}
W = \{x_1, \ldots, x_n\}; \\
\mathsf{while}\ (W \neq \emptyset)\ \{ \\
\qquad \begin{array}{lll} x_i & = & \mathsf{extract}\ W; \\ t & = & f_i\ \mathsf{eval}; \end{array} \\
\qquad \mathsf{if}\ (t \not\sqsubseteq D[x_i])\ \{ \\
\qquad\qquad \begin{array}{lll} D[x_i] & = & D[x_i] \sqcup t; \\ W & = & W \cup I[x_i]; \end{array} \\
\qquad \} \\
\}
\end{array}
$$

wobei :

$$
eval\ \ x_j \ \ = \ \ D[x_j]
$$

Beispiel:

$$\begin{array}{lll} x_1 & \supseteq & \{a\} \cup x_3 \\ x_2 & \supseteq & x_3 \cap \{a, b\} \\ x_3 & \supseteq & x_1 \cup \{c\} \end{array}$$

	I
x_1	$\{x_3\}$
x_2	$\emptyset$
x_3	$\{x_1, x_2\}$

Beispiel:

$$
\begin{array}{lll}
x_1 & \supseteq & \{a\} \cup x_3 \\
x_2 & \supseteq & x_3 \cap \{a, b\} \\
x_3 & \supseteq & x_1 \cup \{c\}
\end{array}
$$

	I
x_1	$\{x_3\}$
x_2	$\emptyset$
x_3	$\{x_1, x_2\}$

$D[x_1]$	$D[x_2]$	$D[x_3]$	W
$\emptyset$	$\emptyset$	$\emptyset$	$\boxed{x_1}, x_2, x_3$
$\{a\}$	$\emptyset$	$\emptyset$	$\boxed{x_2}, x_3$
$\{a\}$	$\emptyset$	$\emptyset$	$\boxed{x_3}$
$\{a\}$	$\emptyset$	$\{a, c\}$	$\boxed{x_1}, x_2$
$\{a, c\}$	$\emptyset$	$\{a, c\}$	$\boxed{x_3}, x_2$
$\{a, c\}$	$\emptyset$	$\{a, c\}$	$\boxed{x_2}$
$\{a, c\}$	$\{a\}$	$\{a, c\}$	$\emptyset$

Theorem

Sei $x_i \sqsupseteq f_i\,(x_1, \ldots, x_n)\;,\quad i = 1, \ldots, n$ ein Constraint-System über dem vollständigen Verband $\mathbb{D}$ der Höhe $h > 0$.

(1) Der Algorithmus terminiert nach maximal $h \cdot N$ Auswertungen rechter Seiten, wobei

$$N = \sum_{i=1}^{n} (1 + \#\,(Dep\ f_i)) \qquad // \quad \text{Größe des Systems} \quad :\text{-})$$

(2) Der Algorithmus liefert eine Lösung. Sind alle f_i monoton, liefert er die kleinste.

Beweis:

Zu (1):

Jede Variable x_i kann nur h mal ihren Wert ändern :-)

Dann wird die Menge $I[x_i]$ zu W hinzu gefügt.

Damit ist die Anzahl an Auswertungen:

$$
\begin{aligned}
&\leq n + \sum_{i=1}^{n} (h \cdot \#\,(I[x_i])) \\
&= n + h \cdot \sum_{i=1}^{n} \#\,(I[x_i]) \\
&= n + h \cdot \sum_{i=1}^{n} \#\,(Dep\ f_i) \\
&\leq h \cdot \sum_{i=1}^{n} (1 + \#\,(Dep\ f_i)) \\
&= h \cdot N
\end{aligned}
$$

Zu (2):

wir betrachten nur die Aussage für monotone f_i .

Sei D_0 die kleinste Lösung. Man zeigt:

- $D_0[x_i] \sqsupseteq D[x_i]$ (zu jedem Zeitpunkt)
- $D[x_i] \not\sqsupseteq f_i \; \mathsf{eval} \Longrightarrow x_i \in W$ (am Ende des Rumpfs)
- Bei Terminierung liefert der Algo eine Lösung :-))

Diskussion:

- Im Beispiel werden tatsächlich weniger Auswertungen rechter Seiten benötigt als bei RR-Iteration :-)
- Der Algo funktioniert auch für nicht-monotone f_i :-)
- Für monotone f_i kann man den Algo vereinfachen:

$$\boxed{D[x_i] = D[x_i] \sqcup t;} \quad \Longrightarrow \quad \boxed{D[x_i] = \qquad t;}$$

- Für Widening ersetzt man:

$$\boxed{D[x_i] = D[x_i] \sqcup t;} \quad \Longrightarrow \quad \boxed{D[x_i] = D[x_i] \sqcup\!\!\!\!\sqcup\, t;}$$

- Für Narrowing ersetzt man:

$$\boxed{D[x_i] = D[x_i] \sqcup t;} \quad \Longrightarrow \quad \boxed{D[x_i] = D[x_i] \sqcap\!\!\!\!\sqcap\, t;}$$

Achtung:

- Der Algorithmus benötigt die Variablen-Abhängigkeiten *Dep* f_i .

  In unseren bisherigen Anwendungen waren die offensichtlich. Das muss nicht immer so sein :-(

- Wir benötigen eine Strategie für extract , die festlegt, welche Variable als nächstes auszuwerten ist.

- Am besten wäre es, wenn wir erst auswerten, dann auf das Ergebnis zugreifen ... :-)

$\Longrightarrow$ rekursive Auswertung ...

Idee:

→ Greifen wir in f_i auf ein x_j zu, werten wir erst rekursiv aus. Dann fügen wir x_i zu $I[x_j]$ hinzu :-)

$$
\begin{array}{lll}
\textsf{eval}\ x_i\ x_j & = & \textsf{solve}\ x_j; \\
 & & I[x_j] = I[x_j] \cup \{x_i\}; \\
 & & D[x_j];
\end{array}
$$

→ Damit die Rekursion nicht unendlich absteigt, verwalten wir die Menge *Stable* von Variablen, für die solve den Wert nachschlägt :-)

Anfangs ist $\mathit{Stable} = \emptyset$ ...

Die Funktion solve :

$$
\begin{array}{lll}
\text{solve } x_i & = & \text{if } (x_i \notin \mathit{Stable}) \; \{ \\
& & \qquad \mathit{Stable} = \mathit{Stable} \cup \{x_i\}; \\
& & \qquad t = f_i \, (\text{eval } \; x_i); \\
& & \qquad \text{if } (t \not\sqsubseteq D[x_i]) \; \{ \\
& & \qquad\qquad W = I[x_i]; \quad I[x_i] = \emptyset; \\
& & \qquad\qquad D[x_i] = D[x_i] \sqcup t; \\
& & \qquad\qquad \mathit{Stable} = \mathit{Stable} \backslash W; \\
& & \qquad\qquad \text{app } \; \text{solve } \; W; \\
& & \qquad \} \\
& & \}
\end{array}
$$

Helmut Seidl, TU München ;-)

Beispiel:

Betrachte unser Standard-Beispiel:

$$\begin{array}{lll} x_1 & \supseteq & \{a\} \cup x_3 \\ x_2 & \supseteq & x_3 \cap \{a, b\} \\ x_3 & \supseteq & x_1 \cup \{c\} \end{array}$$

Dann sieht ein Trace des Fixpunkt-Algorithmus etwa so aus:

solve x_2	eval x_2 x_3	solve x_3	eval x_3 x_1	solve x_1	eval x_1 x_3	solve x_3
						stable!
					$I[x_3] = \{x_1\}$	
					$\Rightarrow \quad \emptyset$	
				$\boxed{D[x_1] = \{a\}}$		
			$I[x_1] = \{x_3\}$			
			$\Rightarrow \quad \{a\}$			
		$\boxed{D[x_3] = \{a, c\}}$				
		$I[x_3] = \emptyset$				
		solve x_1	eval x_1 x_3	solve x_3		
				stable!		
			$I[x_3] = \{x_1\}$			
			$\Rightarrow \quad \{a, c\}$			
		$\boxed{D[x_1] = \{a, c\}}$				
		$I[x_1] = \emptyset$				
		solve x_3	eval x_3 x_1	solve x_1		
				stable!		
			$I[x_1] = \{x_3\}$			
			$\Rightarrow \quad \{a, c\}$			
		$\boxed{\text{ok}}$				
	$I[x_3] = \{x_1, x_2\}$					
	$\Rightarrow \quad \{a, c\}$					
$\boxed{D[x_2] = \{a\}}$						

→ Die Auswertung startet mit einer interessierenden Variable x_i (z.B. dem Wert für *stop*)

→ Es werden automatisch alle Variablen ausgewertet, die x_i beeinflussen :-)

→ Die Anzahl der Auswertungen ist i.a. kleiner als die bei normaler Iteration ;-)

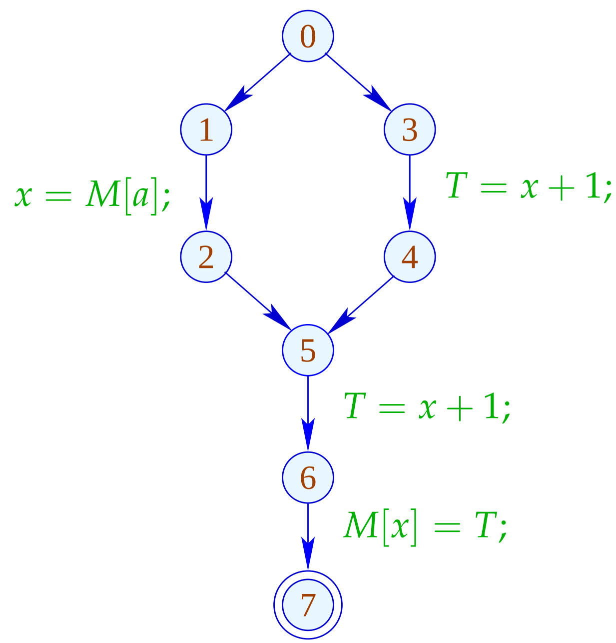
→ Der Algorithmus ist komplizierter, benötigt aber keine Vorberechnung der Variablen-Abhängigkeiten :-))

→ Er funktioniert auch, wenn die Variablen-Abhängigkeiten sich während der Iteration ändern !!!

$\Longrightarrow$ interprozedurale Analyse

1.7 Beseitigung partieller Redundanzen

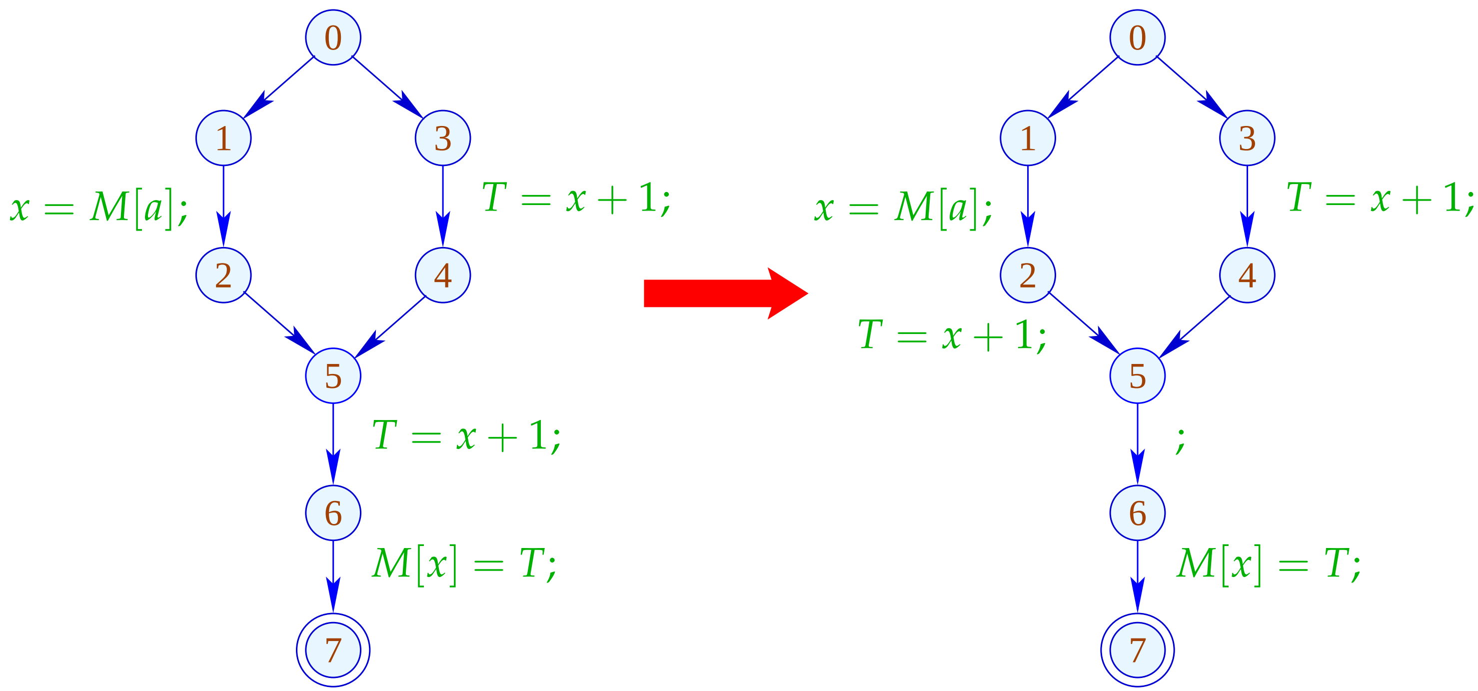
Beispiel:



// $e = x + 1$ wird auf jedem Pfad ausgewertet ...

// leider auf einem Pfad sogar zweimal :-(

Ziel:



Idee:

(1) Wende T1 an, d.h. ersetze jede interessierende Zuweisung x=e; durch: $T_e = e; x = T_e;$

(2) Finde alle Stellen, an denen e sicher berechnet werden kann, ohne die Semantik zu zerstören.

(3) Platziere (konzeptuell) $T_e = e;$ an allen diesen Plätzen.

Beseitige die redundanten Zuweisungen mittels T2.

$\Longrightarrow$ wir benötigen eine neue Analyse :-))

Ein Ausdruck e heißt aktiv (busy) entlang eines Pfads π, falls der Wert von e berechnet wird, bevor eine der Variablen $x \in Vars(e)$ überschrieben wird.

// Rückwärtsanalyse!

e heißt sehr aktiv (very busy) an u, falls e aktiv ist entlang jedes Pfads $\pi : u \to^* stop$.

Ein Ausdruck e heißt aktiv (busy) entlang eines Pfads π, falls der Wert von e berechnet wird, bevor eine der Variablen $x \in Vars(e)$ überschrieben wird.

// Rückwärtsanalyse!

e heißt sehr aktiv (very busy) an u, falls e aktiv ist entlang jedes Pfads $\pi : u \to^* stop$.

Entsprechend benötigen wir:

$$\mathcal{B}[u] = \bigcap \{ [\![\pi]\!]^\sharp\, \emptyset \mid \pi : u \to^* stop \}$$

wobei für $\pi = k_1 \ldots k_m$:

$$[\![\pi]\!]^\sharp = [\![k_1]\!]^\sharp \circ \ldots \circ [\![k_m]\!]^\sharp$$

Unser vollständiger Verband ist:

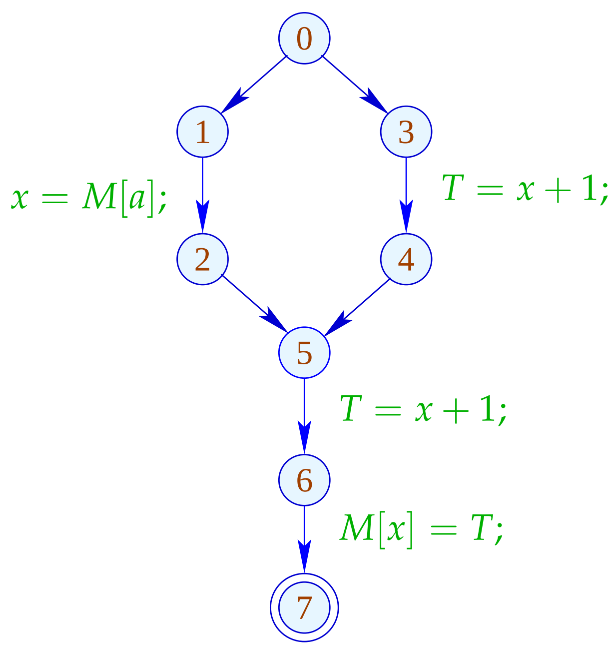
$$\mathbb{B} = 2^{Expr\backslash Vars} \qquad \text{mit} \qquad \sqsubseteq \; = \; \supseteq$$

Der Effekt $[\![k]\!]^\sharp$ einer Kante $k = (u, lab, v)$ hängt nur von lab ab, d.h. $[\![k]\!]^\sharp = [\![lab]\!]^\sharp$ wobei:

$$\begin{array}{lllll}
[\![;]\!]^\sharp\, B & = & B & & \\
[\![Pos(e)]\!]^\sharp\, B & = & [\![Neg(e)]\!]^\sharp\, B & = & B \cup \{e\} \\
[\![T_e = e;]\!]^\sharp\, B & = & (B \backslash Expr_{T_e}) \cup \{e\} & & \\
[\![x = T;]\!]^\sharp\, B & = & B \backslash Expr_x & & \\
[\![x = M[R];]\!]^\sharp\, B & = & B \backslash Expr_x & & \\
[\![M[R] = x;]\!]^\sharp\, B & = & B & &
\end{array}$$

Die Kanten-Effekte sind sämtlich distributiv. Deshalb liefert die kleinste Lösung des Constraint-Systems exakt den MOP :-)

Beispiel:



7	$\emptyset$
6	$\emptyset$
5	$\{x + 1\}$
4	$\{x + 1\}$
3	$\{x + 1\}$
2	$\{x + 1\}$
1	$\emptyset$
0	$\emptyset$

Beachte:

- Im Beispiel enthalten die $\mathcal{B}[u]$ maximal ein Element.
- Enthält $\mathcal{B}[u]$ mehrere Ausdrücke $e_1 \neq e_2$, sind diese unahängig, d.h. $T_{e_1} \not\in Vars\,(e_2)$:-)
- Unabhängige Ausdrücke können in beliebiger Reihenfolge berechnet werden :-))

Beachte:

- Im Beispiel enthalten die $\mathcal{B}[u]$ maximal ein Element.
- Enthält $\mathcal{B}[u]$ mehrere Ausdrücke $e_1 \neq e_2$, können diese stets in einer Reihenfolge $e_1 \longrightarrow e_2$ ausgewertet werden, so dass $T_{e_2} \not\in \mathit{Vars}\,(e_1)$:-)

Beweis der Anordbarkeit:

$\longrightarrow$ Wir zeigen Beh. für $[\![\pi]\!]^\sharp\,\emptyset$. Die Beh. für u folgt, da die Eigenschaft unter $\cap$ abgeschlossen ist.

$\longrightarrow$ Induktion über die Länge von π.

$\boxed{\pi = \epsilon}$ $[\![\pi]\!]^\sharp\,\emptyset = [\![\epsilon]\!]^\sharp\,\emptyset = \emptyset$:-)

$\boxed{\pi = k\;\pi'}$ Kanten-Effekte erhalten die Anordbarkeit :-))

Ein u heißt sicher für e , sofern $e \in \mathcal{A}[u] \cup \mathcal{B}[u]$; d.h. e ist entweder verfügbar oder sehr aktiv.

Ist u sicher, können wir dort e gefahrlos berechnen :-)

Idee:

- Wir berechnen e zum frühest möglichen Zeitpunkt :-)
- Wir platzieren die Berechnung von e am Ende von $k = (u, lab, v)$ falls:

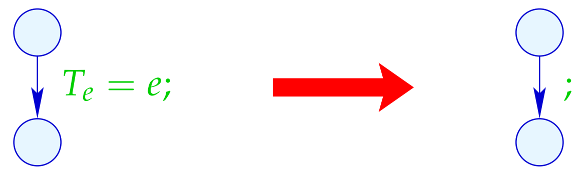
  $\longrightarrow$ $e \in \mathcal{B}[v]$ sowie

  $\longrightarrow$ $e \not\in [\![lab]\!]^\sharp\,(\mathcal{A}[u])$ (nicht verfügbar entlang k) und

  $\longrightarrow$ $e \not\in [\![lab]\!]^\sharp\,(\mathcal{B}[u])$ (auch nicht nach Transformation)

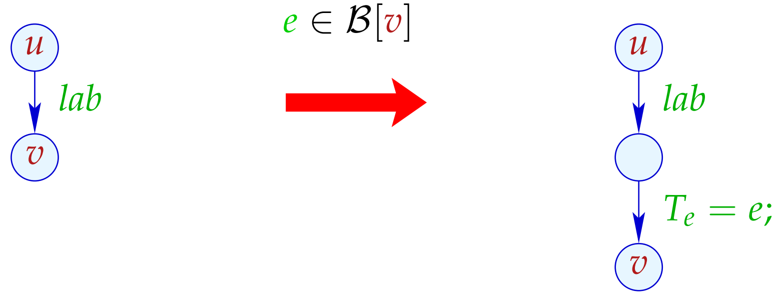
- Weil alle $e \in \mathcal{B}[v]$ anordbar sind, betrachten wir die Transformation für jedes e gesondert:

Transformation 6.1:

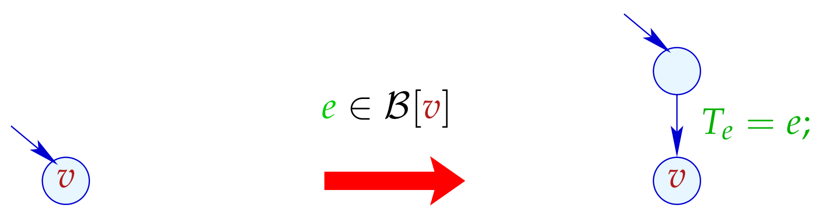


Transformation 6.2:

$$e \notin [\![lab]\!]^\sharp(\mathcal{B}[u] \cup \mathcal{A}[u])$$
$$e \in \mathcal{B}[v]$$



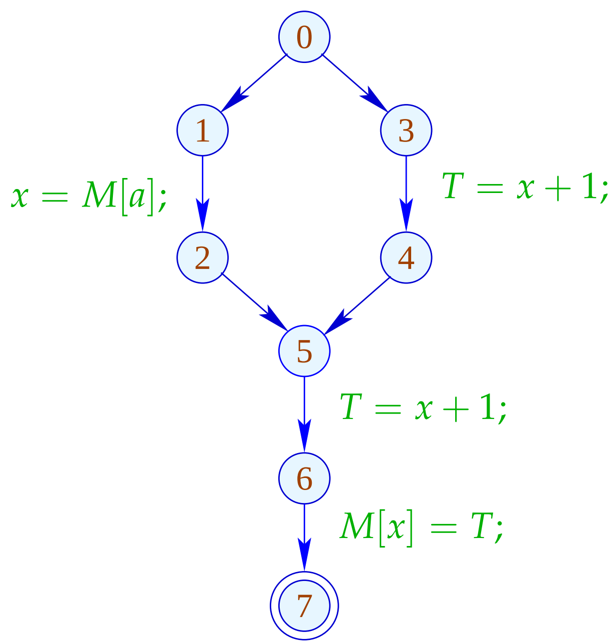
$$e \in \mathcal{B}[v]$$



Bernhard Steffen, Dortmund

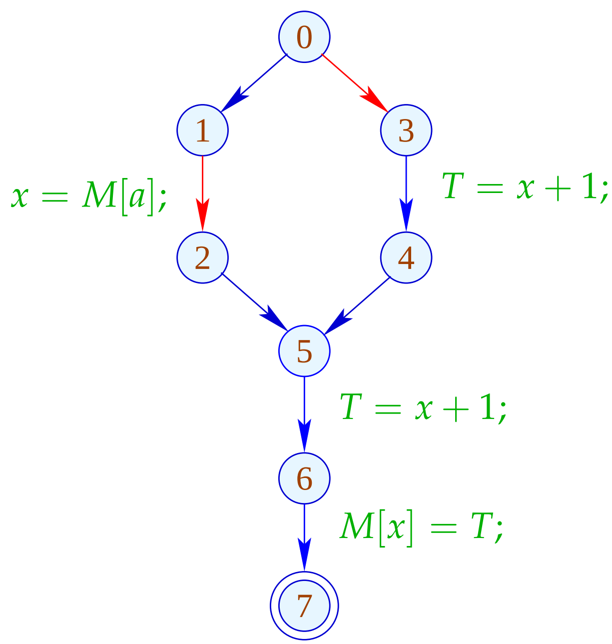
Jens Knoop, Wien

Im Beispiel:



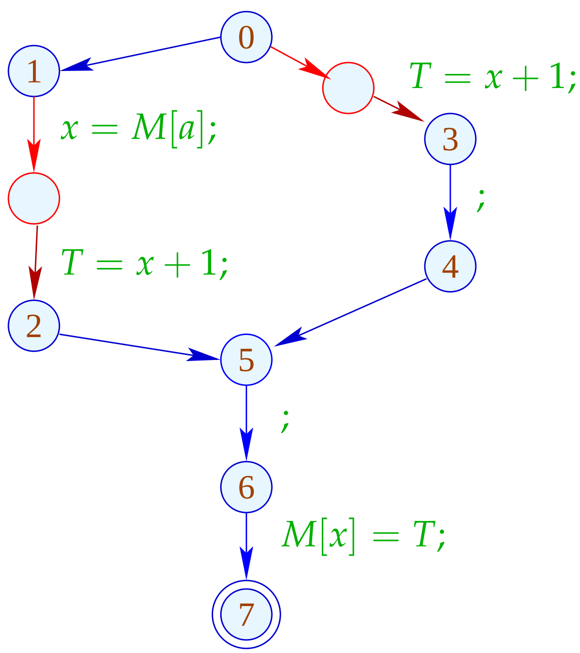
	$\mathcal{A}$	$\mathcal{B}$
0	$\emptyset$	$\emptyset$
1	$\emptyset$	$\emptyset$
2	$\emptyset$	$\{x+1\}$
3	$\emptyset$	$\{x+1\}$
4	$\{x+1\}$	$\{x+1\}$
5	$\emptyset$	$\{x+1\}$
6	$\{x+1\}$	$\emptyset$
7	$\{x+1\}$	$\emptyset$

Im Beispiel:



	$\mathcal{A}$	$\mathcal{B}$
0	$\emptyset$	$\emptyset$
1	$\emptyset$	$\emptyset$
2	$\emptyset$	$\{x+1\}$
3	$\emptyset$	$\{x+1\}$
4	$\{x+1\}$	$\{x+1\}$
5	$\emptyset$	$\{x+1\}$
6	$\{x+1\}$	$\emptyset$
7	$\{x+1\}$	$\emptyset$

Im Beispiel:



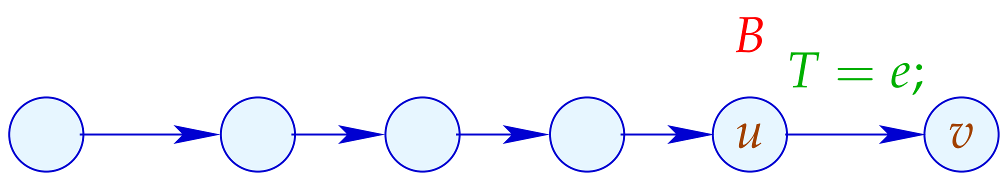
	$\mathcal{A}$	$\mathcal{B}$
0	$\emptyset$	$\emptyset$
1	$\emptyset$	$\emptyset$
2	$\emptyset$	$\{x+1\}$
3	$\emptyset$	$\{x+1\}$
4	$\{x+1\}$	$\{x+1\}$
5	$\emptyset$	$\{x+1\}$
6	$\{x+1\}$	$\emptyset$
7	$\{x+1\}$	$\emptyset$

Zur Korrektheit:

Sei $\pi = \pi' \, k$ ein Pfad mit $k = (u, T = e;, v)$.

Dann gilt: $e \in \mathcal{B}[u]$ da u nur eine ausgehende Kante hat :-)

Wir haben:

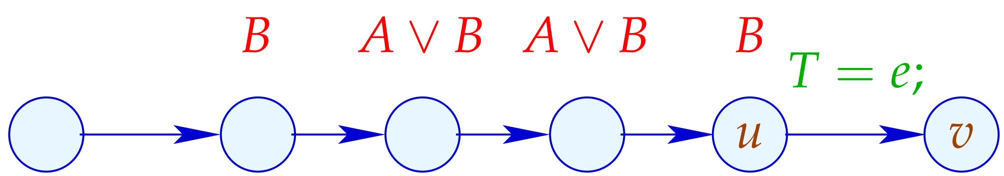


Zur Korrektheit:

Sei $\pi = \pi' \, k$ ein Pfad mit $k = (u, T = e;, v)$.

Dann gilt: $e \in \mathcal{B}[u]$ da u nur eine ausgehende Kante hat :-)

Wir haben:

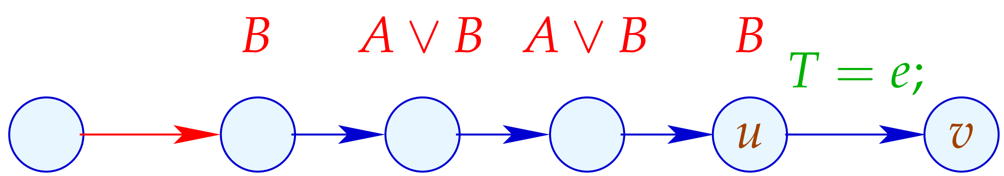


Zur Korrektheit:

Sei $\pi = \pi' \, k$ ein Pfad mit $k = (u, T = e;, v)$.

Dann gilt: $e \in \mathcal{B}[u]$ da u nur eine ausgehende Kante hat :-)

Wir haben:

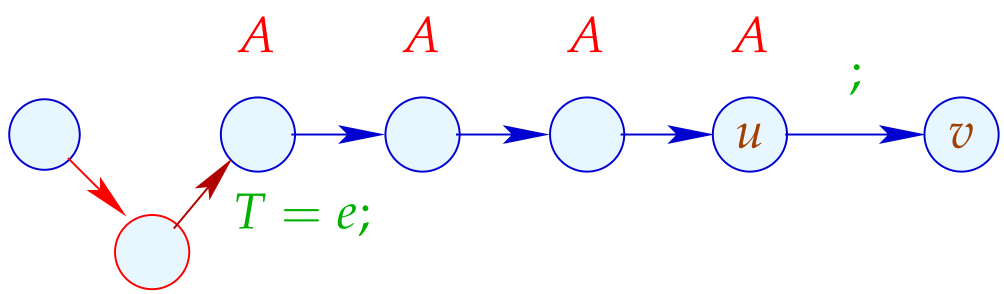


Zur Korrektheit:

Sei $\pi = \pi'\, k$ ein Pfad mit $k = (u, T = e;, v)$.

Dann gilt: $e \in \mathcal{B}[u]$ da u nur eine ausgehende Kante hat :-)

Wir haben:



Wir schließen:

- Überall, wo wir $T = e;$ gestrichen haben, ist e verfügbar :-)

  $\Longrightarrow$ Korrektheit der Transformation

- Jedem $T = e;$, das wir in einen Pfad einfügen, entspricht ein $T = e;$, das wir gestrichen haben :-))

  $\Longrightarrow$ Nicht-Verschlechterung der Transformation

1.8 Anwendung: Schleifen-invarianter Code

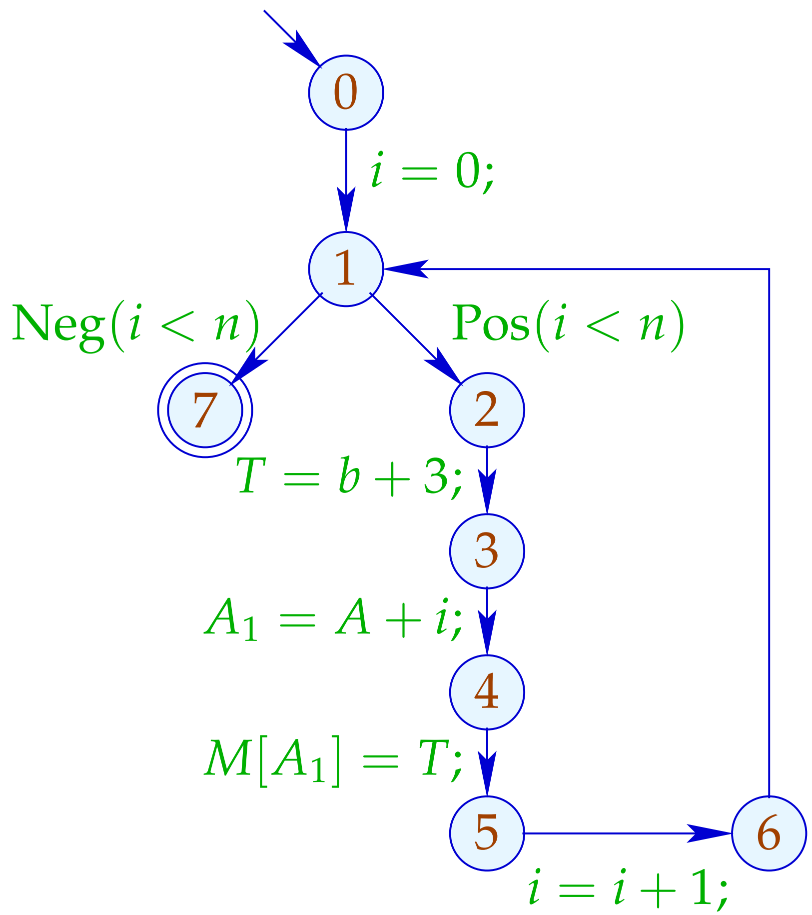
Beispiel:

$$\text{for } (i = 0; i < n; i{+}{+}) \\ a[i] = b + 3;$$

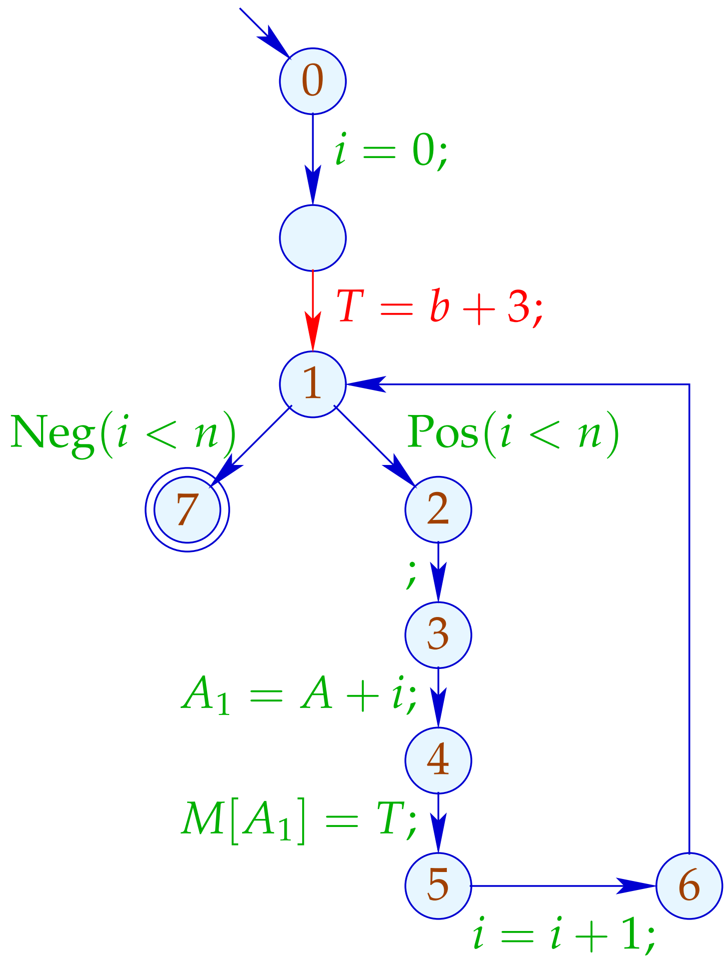
$//$ Der Ausdruck $b + 3$ wird in jeder Iteration berechnet :-(

$//$ Das wollen wir vermeiden :-)

Der Kontrollfluss-Graph:

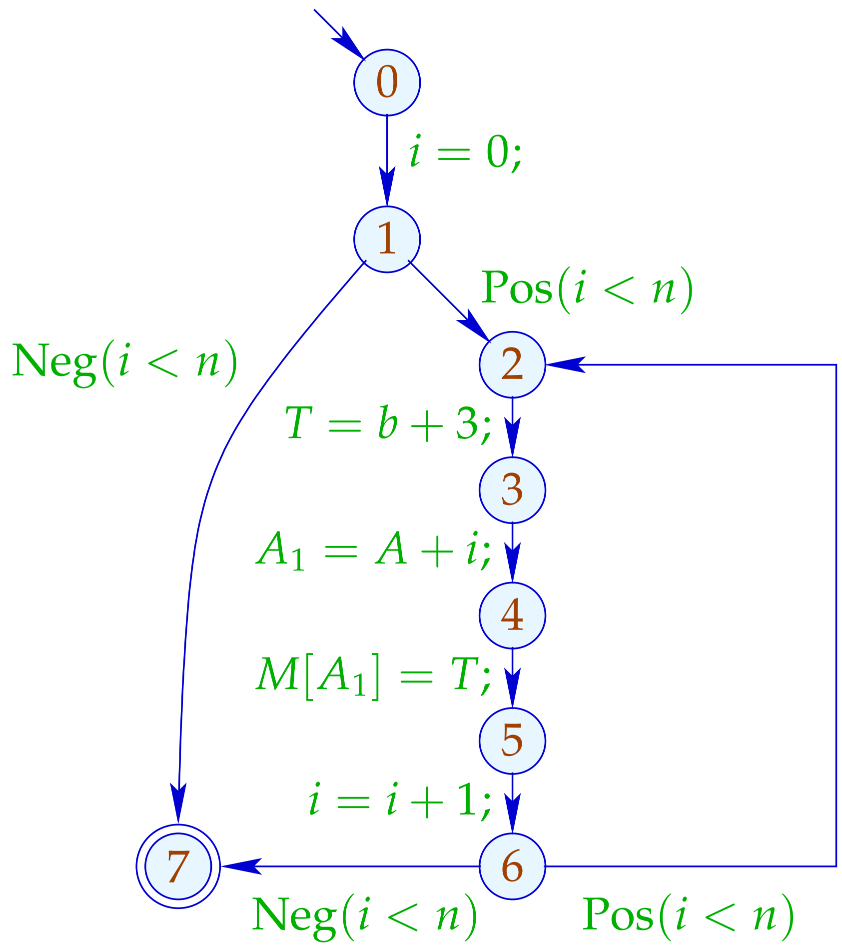


Achtung: $T = b + 3;$ darf nicht vor der Schleife stehen :

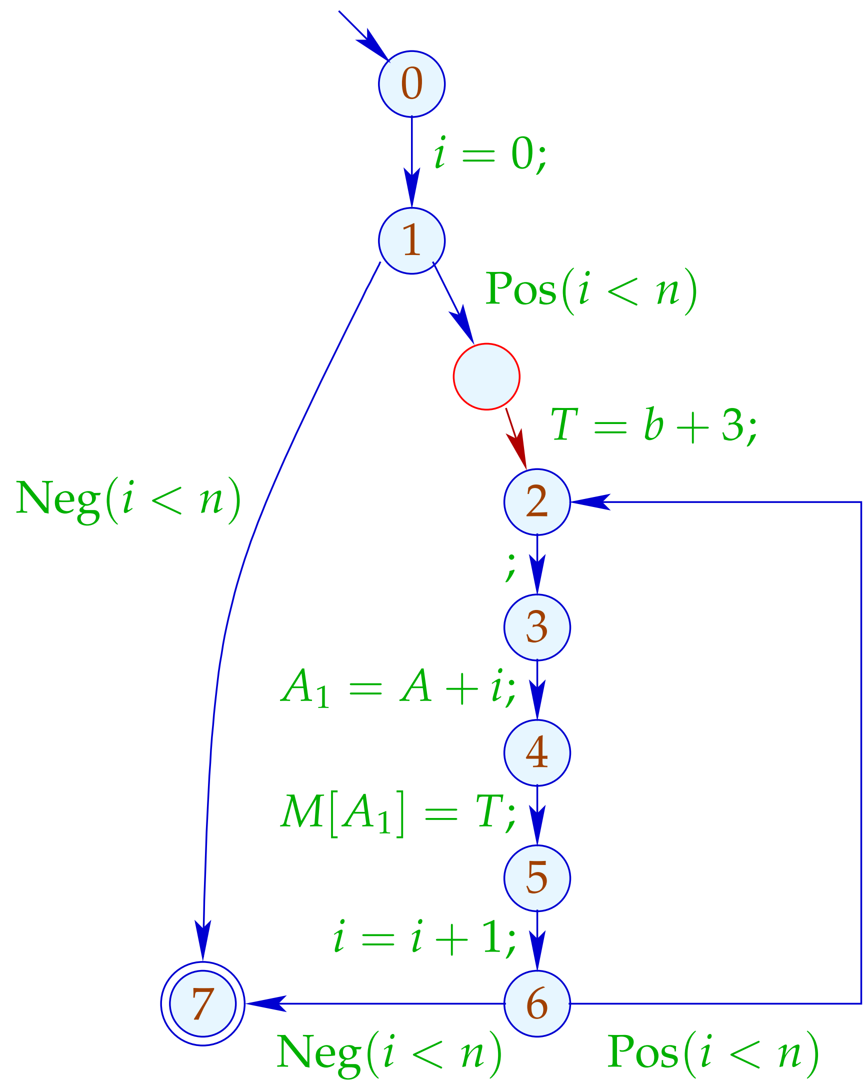


$\Longrightarrow$ Es gibt keinen guten Platz für $T = b + 3;$:-(

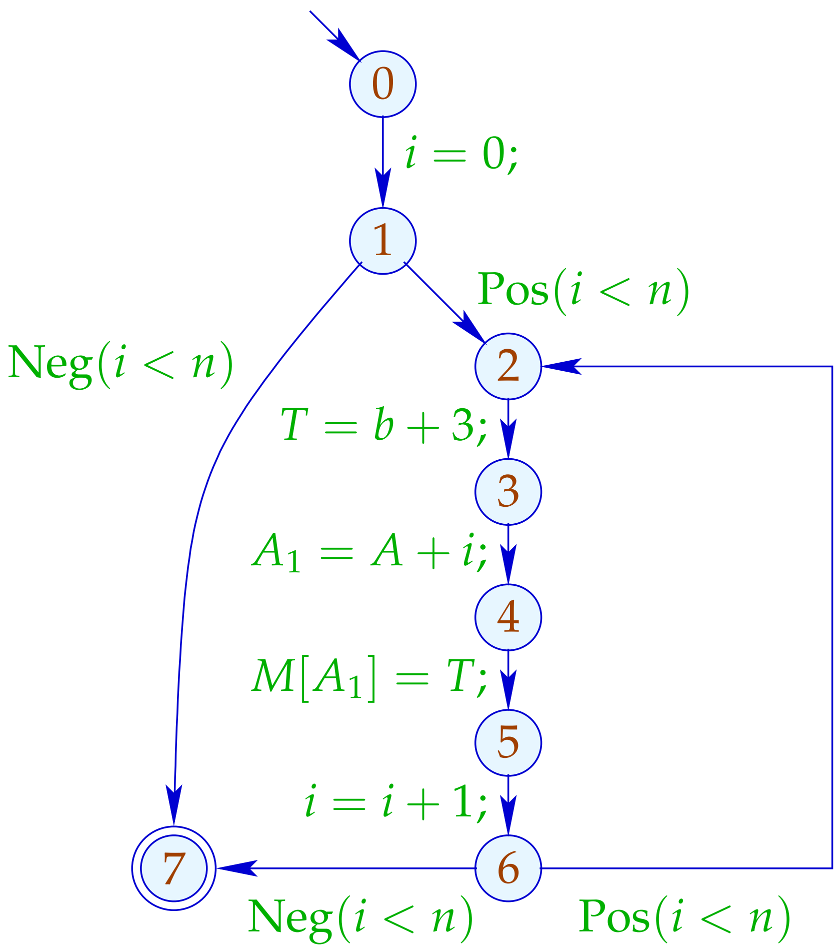
Idee: Transformiere in eine do-while-Schleife ...



... jetzt gibt es eine Stelle für $T = e;$:-)

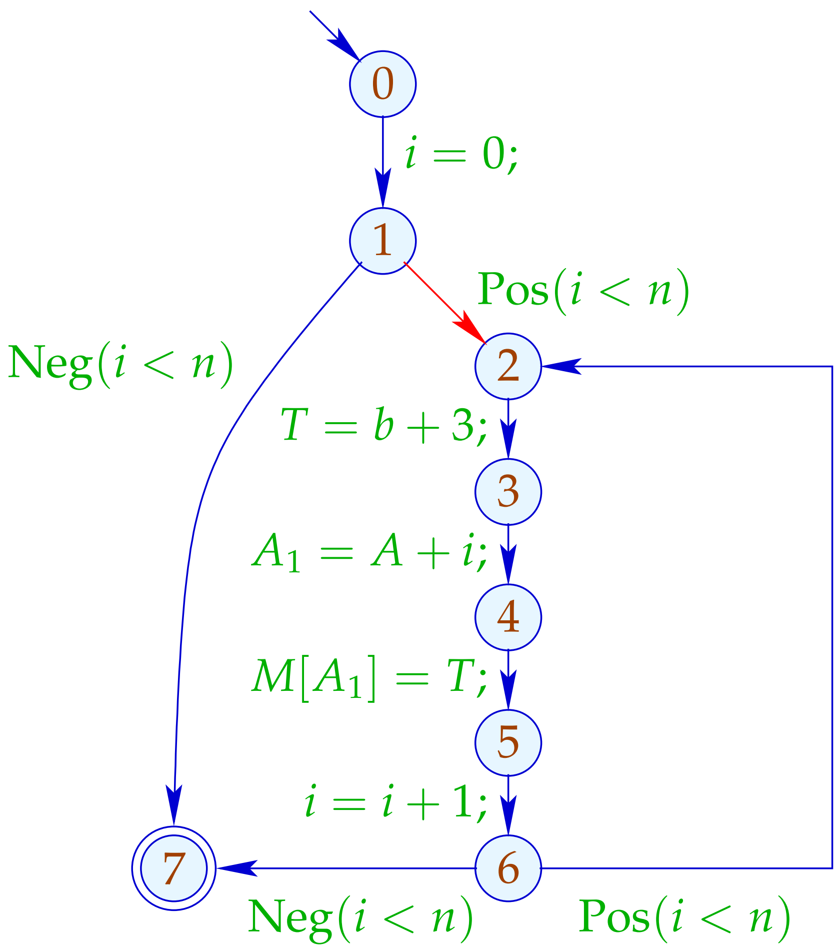


Anwendung von T6 (PRE) :



	$\mathcal{A}$	$\mathcal{B}$
0	$\emptyset$	$\emptyset$
1	$\emptyset$	$\emptyset$
2	$\emptyset$	$\{b+3\}$
3	$\{b+3\}$	$\emptyset$
4	$\{b+3\}$	$\emptyset$
5	$\{b+3\}$	$\emptyset$
6	$\{b+3\}$	$\emptyset$
6	$\emptyset$	$\emptyset$
7	$\emptyset$	$\emptyset$

Anwendung von T6 (PRE) :



	$\mathcal{A}$	$\mathcal{B}$
0	$\emptyset$	$\emptyset$
1	$\emptyset$	$\emptyset$
2	$\emptyset$	$\{b+3\}$
3	$\{b+3\}$	$\emptyset$
4	$\{b+3\}$	$\emptyset$
5	$\{b+3\}$	$\emptyset$
6	$\{b+3\}$	$\emptyset$
6	$\emptyset$	$\emptyset$
7	$\emptyset$	$\emptyset$

Fazit:

- Beseitigung partieller Redundanzen kann loop-invarianten Code aus Schleifen heraus schieben :-))
- Das funktioniert nur für do-while-Scheifen :-(
- Um andere Scheifen zu optimieren, wandeln wir sie in do-while-Scheifen um:

$$\textsf{while}\ (b)\ \mathit{stmt} \quad \Longrightarrow \quad \begin{array}{l} \textsf{if}\ (b) \\ \qquad \textsf{do}\ \mathit{stmt} \\ \qquad \textsf{while}\ (b); \end{array}$$

$\Longrightarrow$ Schleifen-Rotation

Problem:

Haben wir das Quell-Programm nicht (mehr) zur Verfügung, müssen wir nachträglich die Schleifen (-köpfe) identifizieren ;-)

$\Longrightarrow$ Prädominatoren

u prädominiert v , falls jeder Pfad $\pi : \mathit{start} \rightarrow^* v$ Knoten u enthält. Wir schreiben: $u \Rightarrow v$.

"$\Rightarrow$" ist reflexiv, transitiv und anti-symmetrisch :-)

Berechnung:

Wir sammeln die Knoten entlang Pfaden auf mithilfe der Analyse:

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

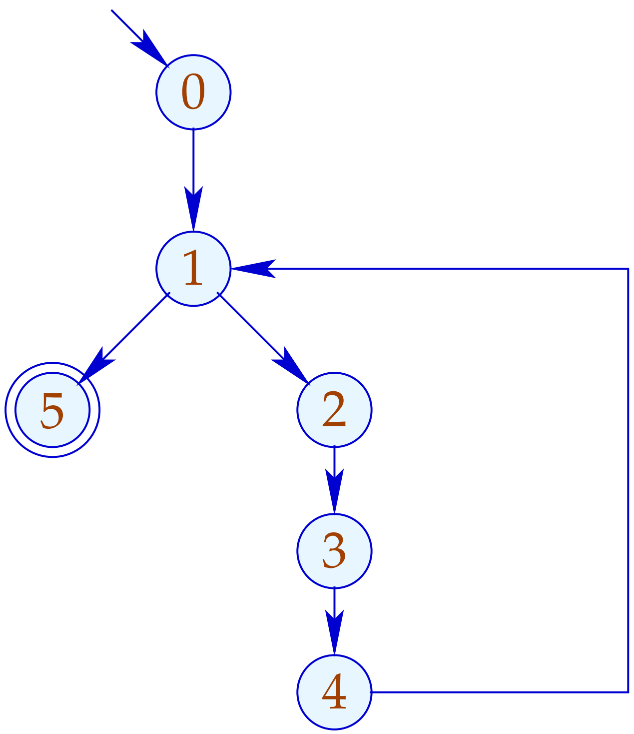
$$[\![(\_,\_,v)]\!]^\sharp \, P \quad = \quad P \cup \{v\}$$

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

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

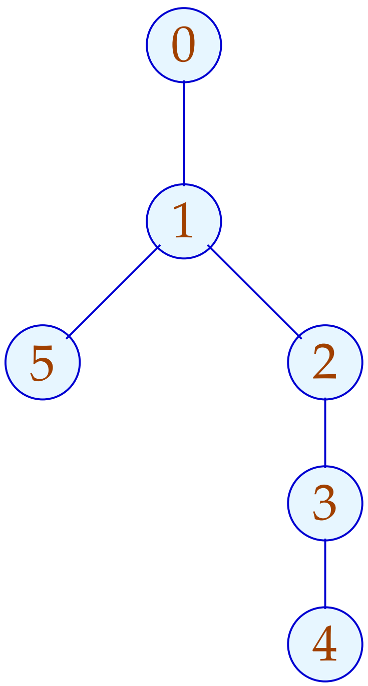
Da die $[\![k]\!]^\sharp$ distributiv sind, können wir die $\mathcal{P}[v]$ mithilfe von Fixpunkt-Iteration berechnen :-)

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.

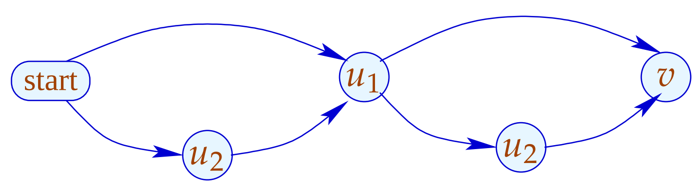
Gälte $u_1 \Rightarrow u_2$, wäre u_1 nicht unmittelbar.

Folglich müssen u_1, u_2 unvergleichbar sein :-)

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

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

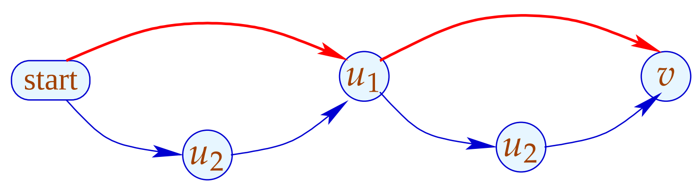
Sind u_1, u_2 aber unvergleichbar, gibt es einen Pfad: $start \rightarrow^* v$ ohne u_2 :



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.

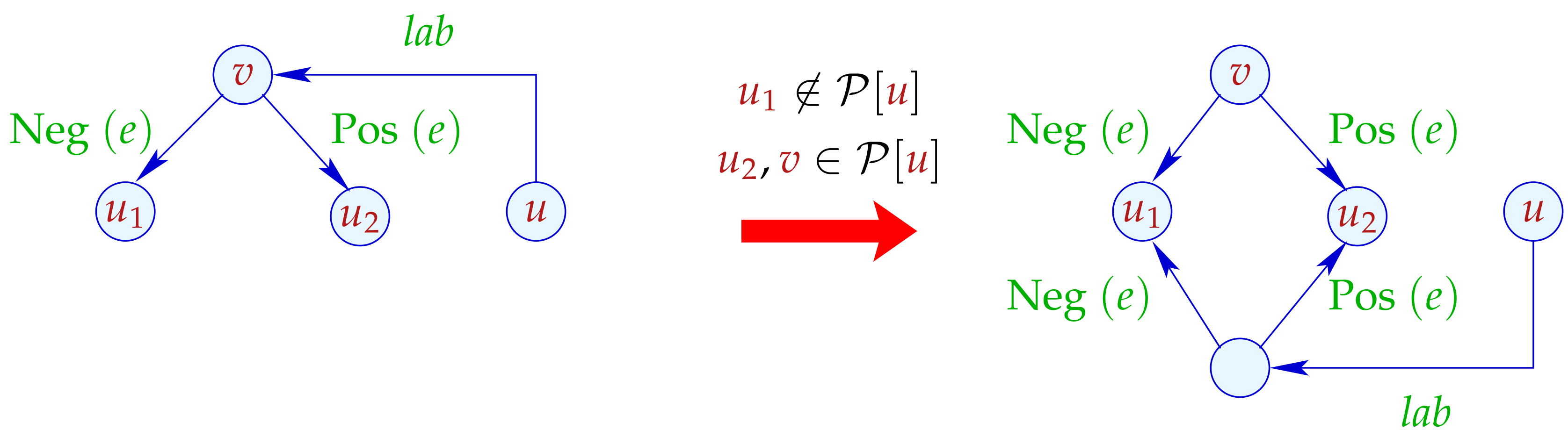
Einen Rücksprung vom Ende u zum Schleifenkopf v erkennt man daran, dass

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

:-)

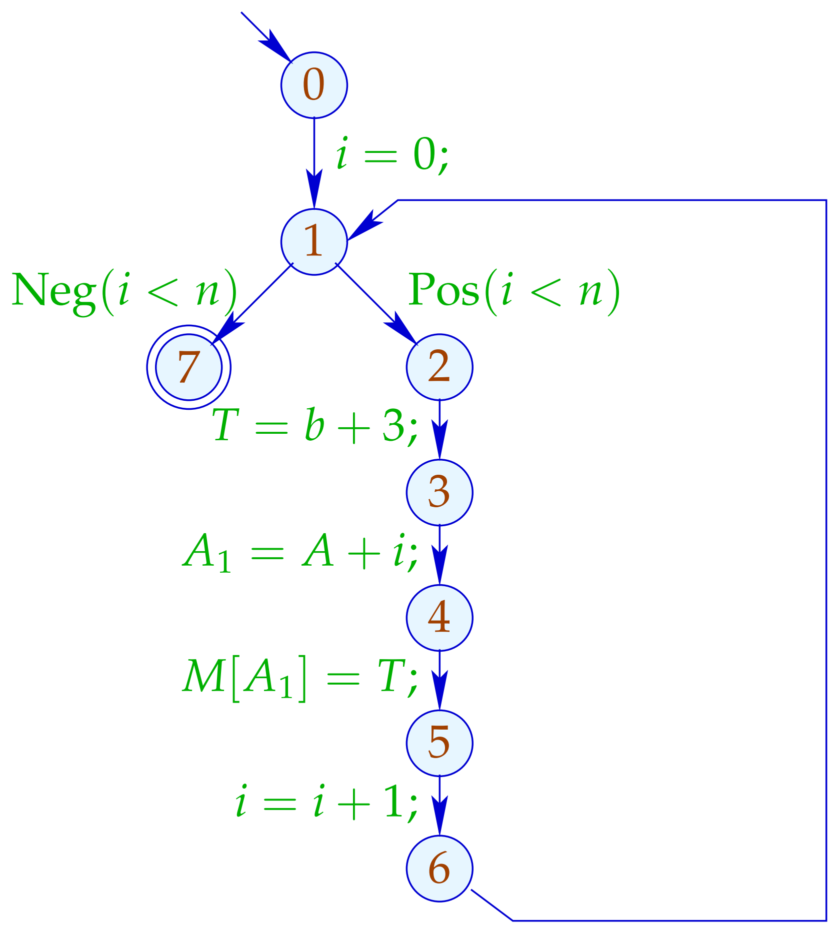
Damit definieren wir:

Transformation 7:

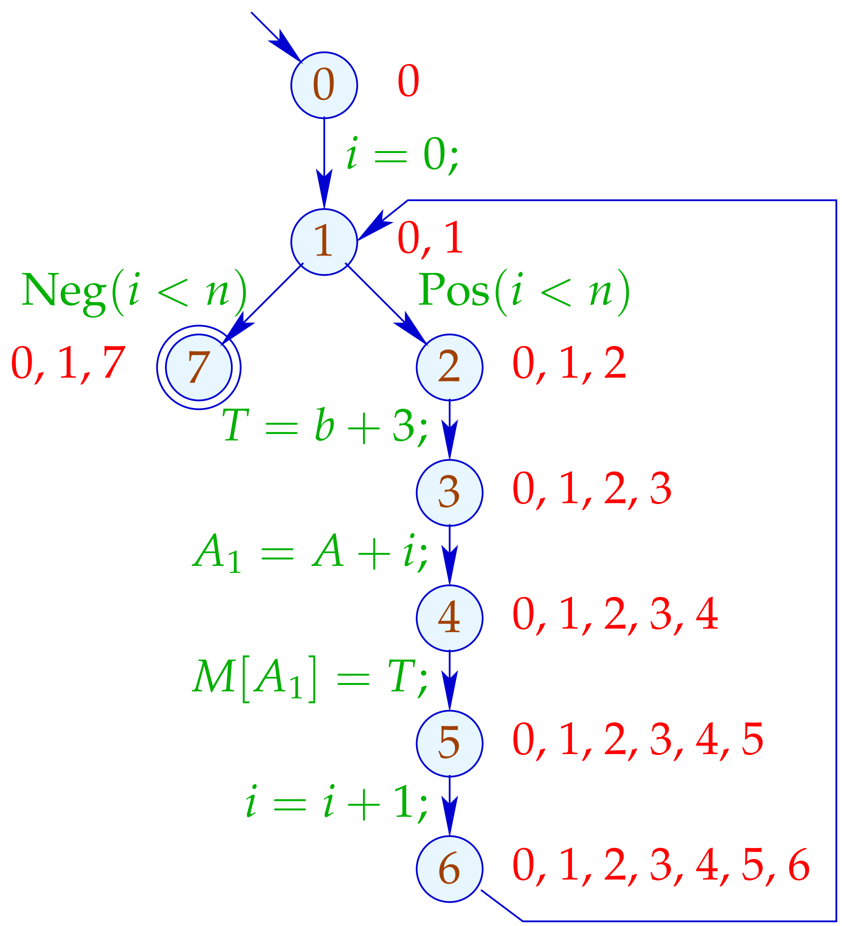


Wir duplizieren den Eintritts-Test an alle Rücksprung-Stellen :-)

... im Beispiel:

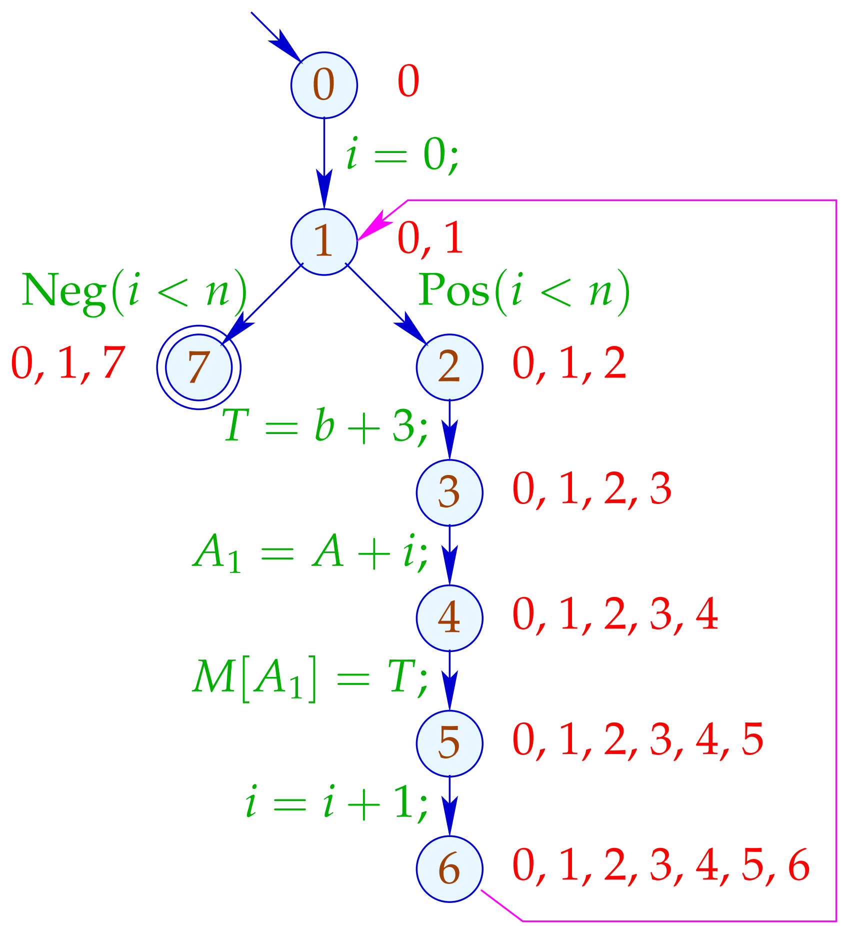


... im Beispiel:



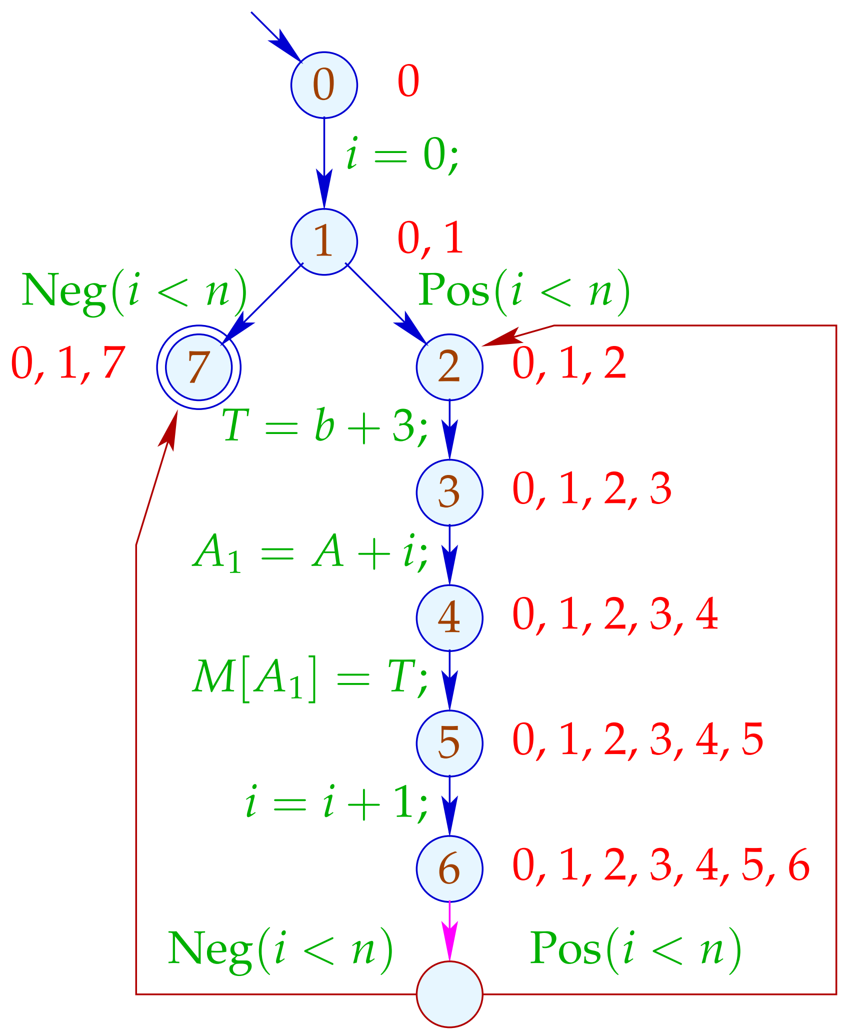
0
0
$i = 0;$
1
0, 1
Neg$(i < n)$
Pos$(i < n)$
7
0, 1, 7
2
0, 1, 2
$T = b + 3;$
3
0, 1, 2, 3
$A_1 = A + i;$
4
0, 1, 2, 3, 4
$M[A_1] = T;$
5
0, 1, 2, 3, 4, 5
$i = i + 1;$
6
0, 1, 2, 3, 4, 5, 6

... im Beispiel:



0
0
$i = 0;$
1
0, 1
$\text{Neg}(i < n)$
$\text{Pos}(i < n)$
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

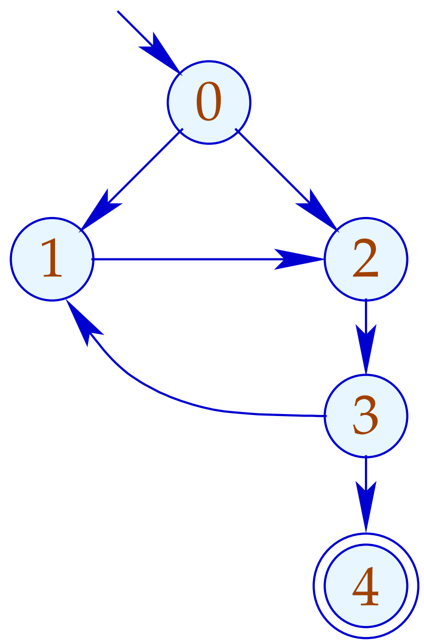
... im Beispiel:



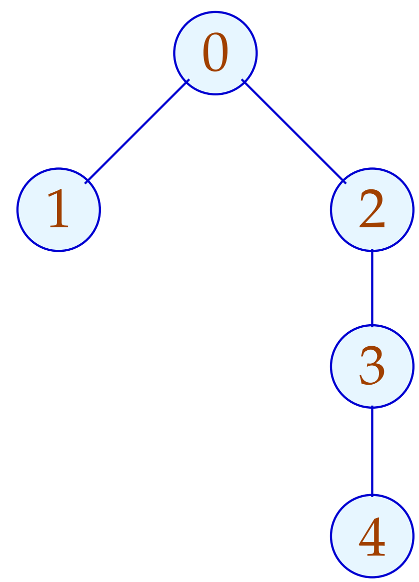
0
0
$i = 0;$
1
0, 1
$\mathrm{Neg}(i < n)$
$\mathrm{Pos}(i < n)$
0, 1, 7
7
2
0, 1, 2
$T = b + 3;$
3
0, 1, 2, 3
$A_1 = A + i;$
4
0, 1, 2, 3, 4
$M[A_1] = T;$
5
0, 1, 2, 3, 4, 5
$i = i + 1;$
6
0, 1, 2, 3, 4, 5, 6
$\mathrm{Neg}(i < n)$
$\mathrm{Pos}(i < n)$

Achtung:

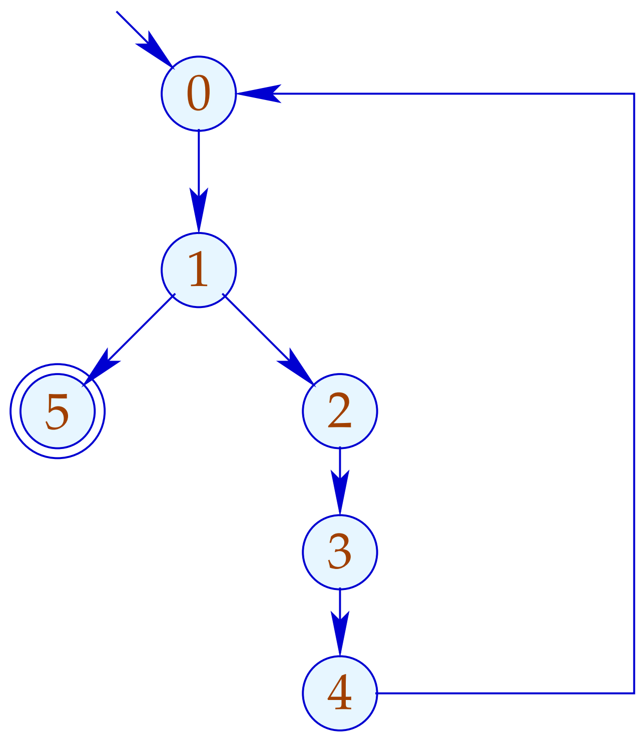
Es gibt ungewöhnliche Schleifen, die so nicht rotiert werden:

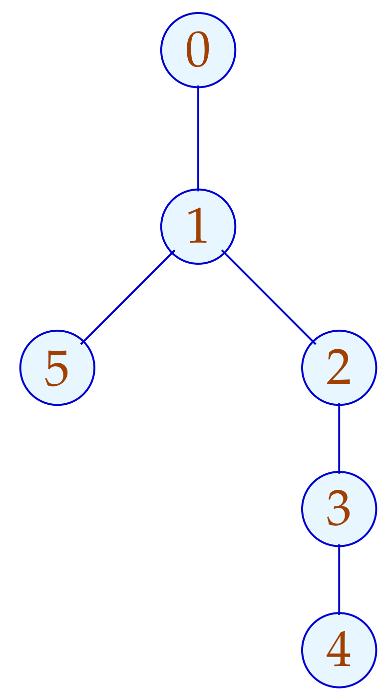


Prädominatoren:



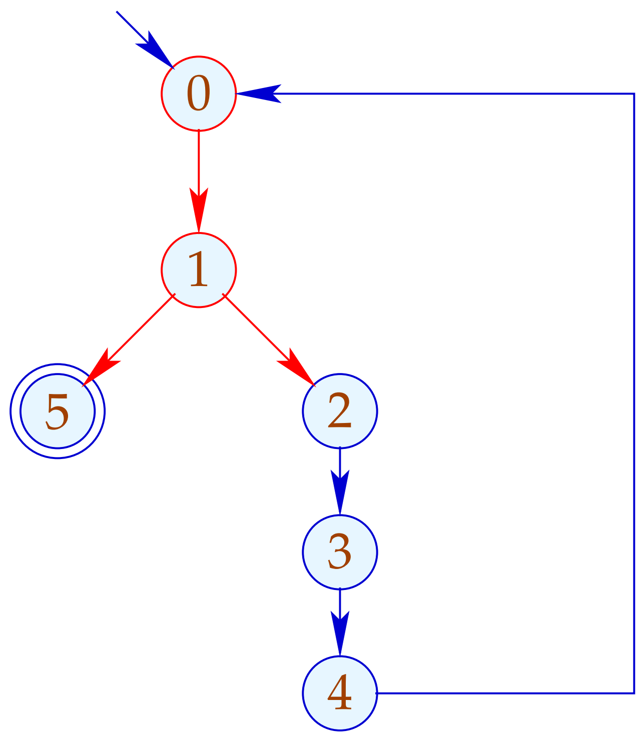
... leider aber auch gewöhnliche, die nicht rotiert werden:

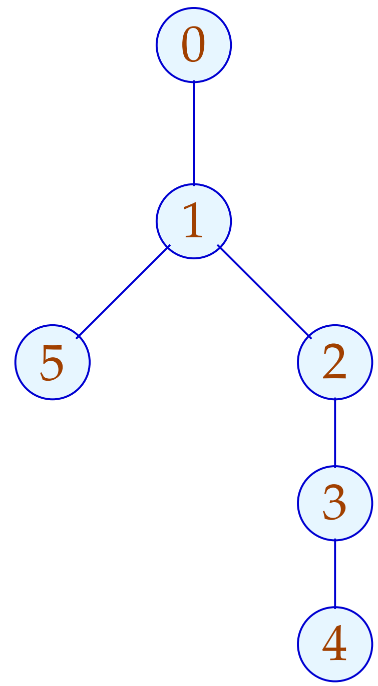




Hier müsste man den ganzen Pfad zwischen Rücksprung und Bedingung duplizieren :-(

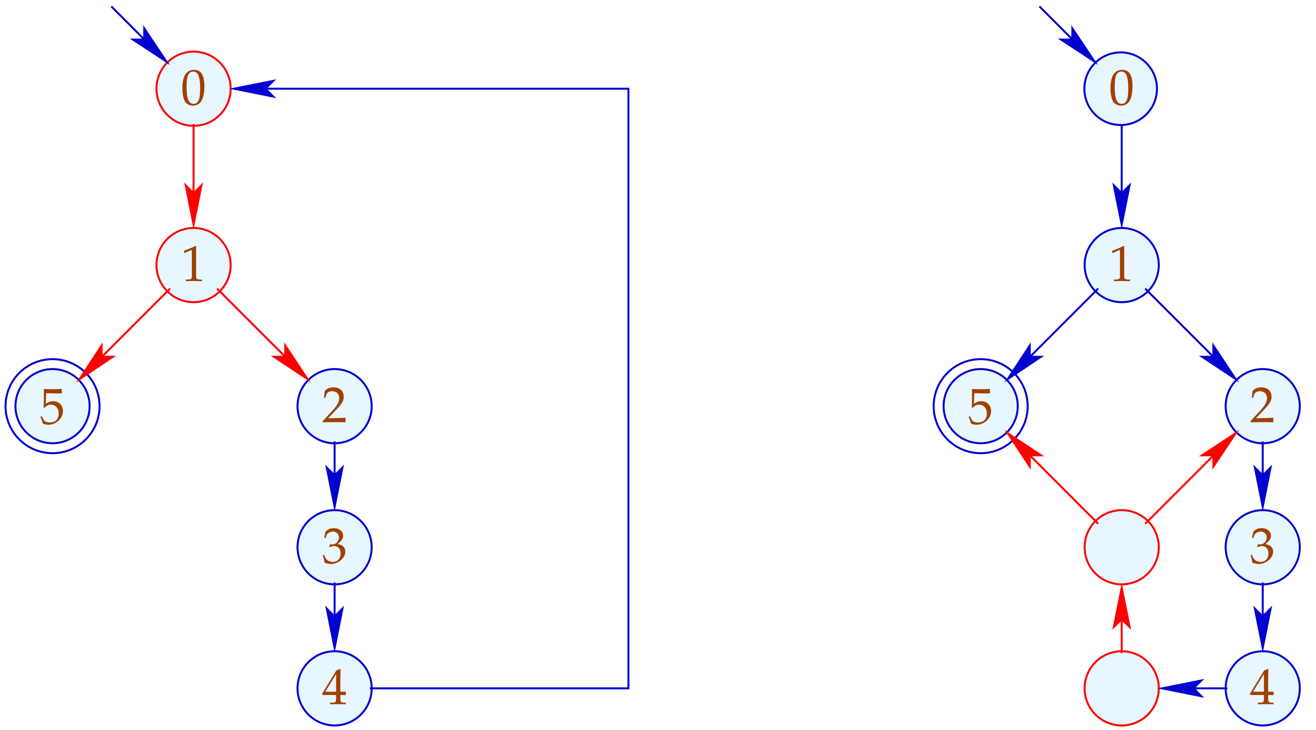
... leider aber auch gewöhnliche, die nicht rotiert werden:





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

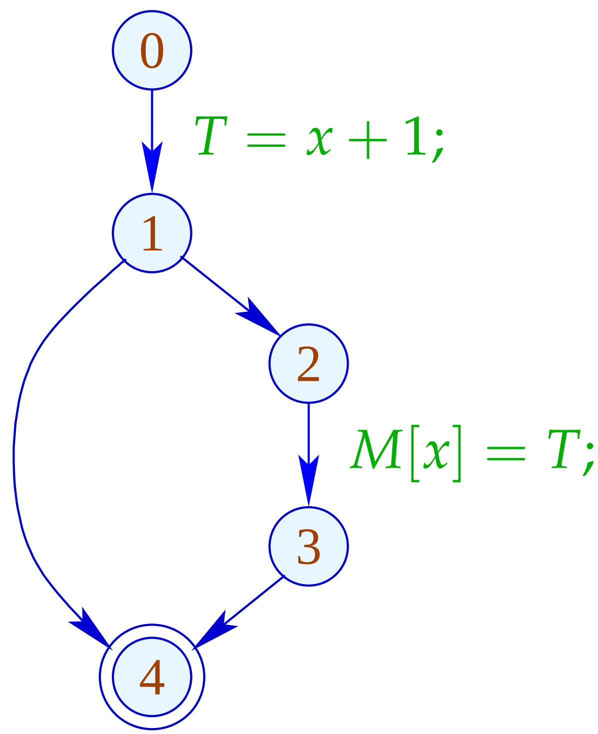
... leider aber auch gewöhnliche, die nicht rotiert werden:



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

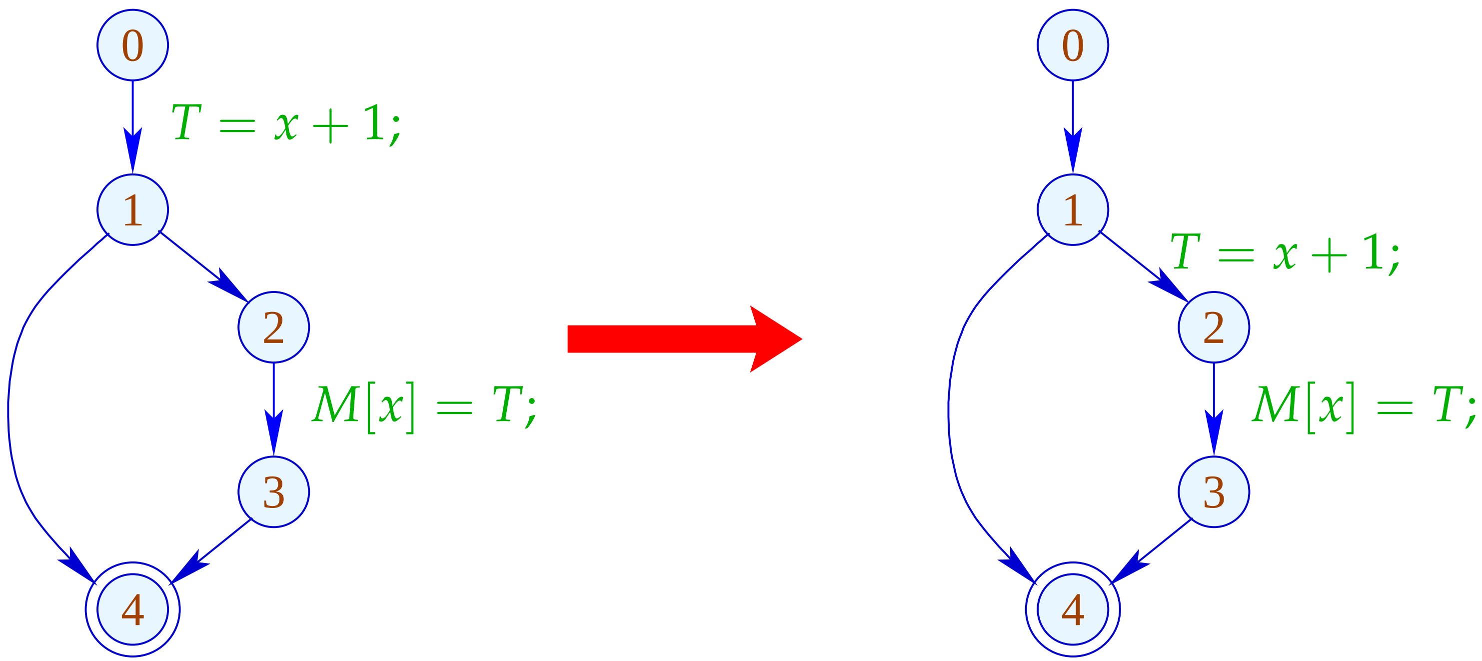
1.9 Beseitigung partiell toten Codes

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:

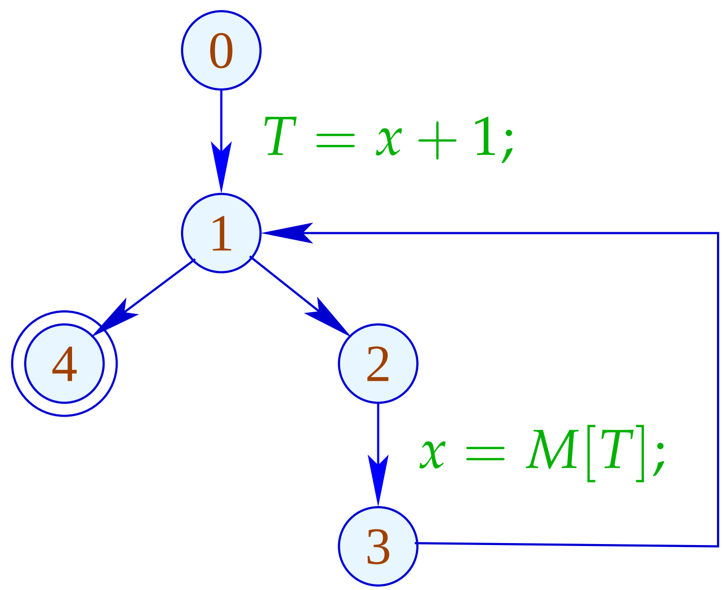
1. $e \in \mathcal{A}[u]$; und
2. $e \notin \mathcal{A}[v] \quad \vee \quad T_e \in Use(lab)$

Dabei ist:

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

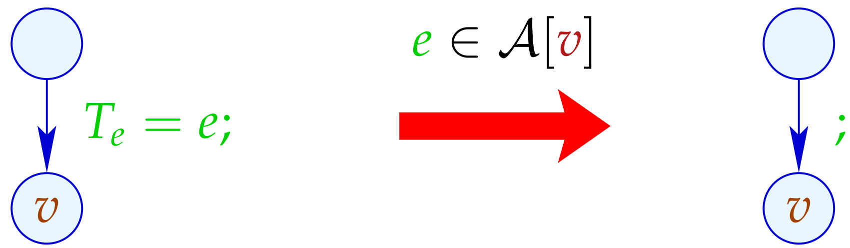
Achtung:

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

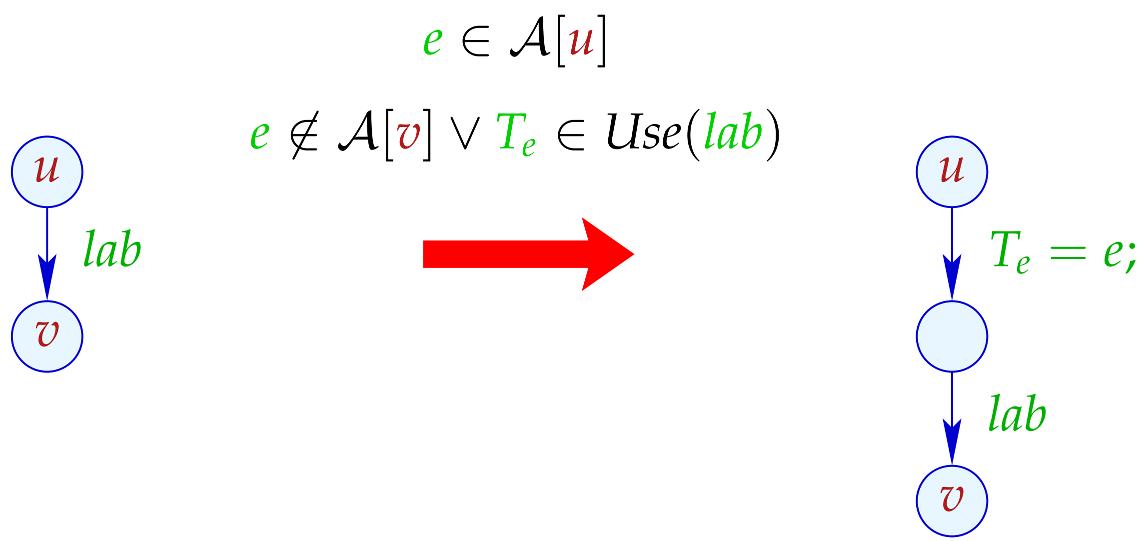


Offenbar ist $x + 1$ nicht an 1 verfügbar, darf also nicht verschoben werden !!!

Transformation 8.1:



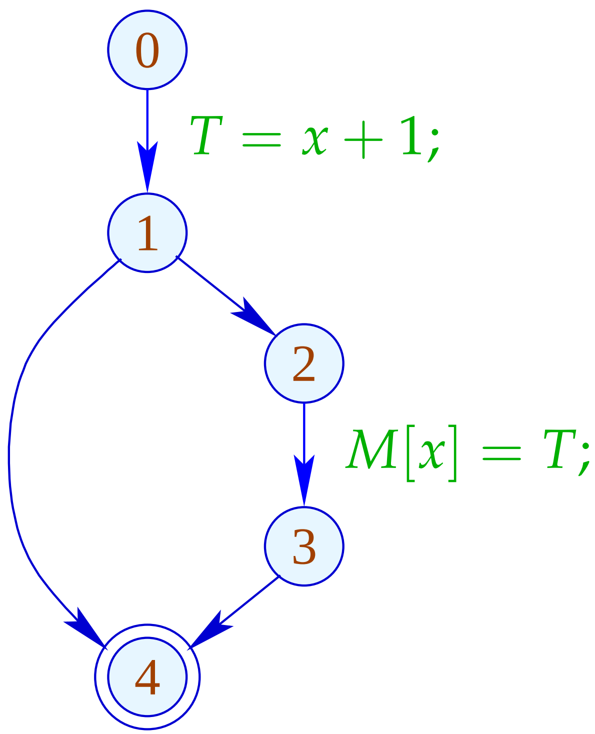
Transformation 8.2:



Beachte:

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

Im Beispiel beseitigt sie den partiell toten Code:

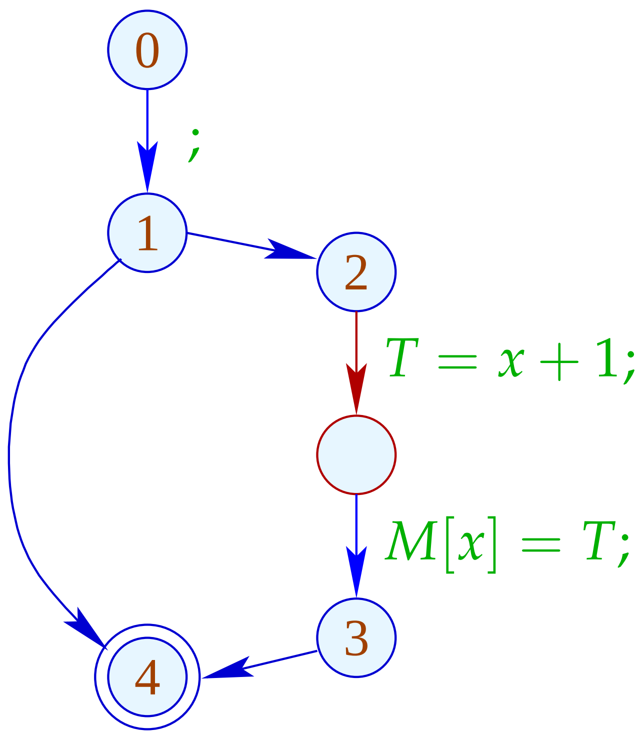


	$\mathcal{A}$
0	$\emptyset$
1	$\{x + 1\}$
2	$\{x + 1\}$
3	$\{x + 1\}$
4	$\{x + 1\}$

Beachte:

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

Im Beispiel beseitigt sie den partiell toten Code:



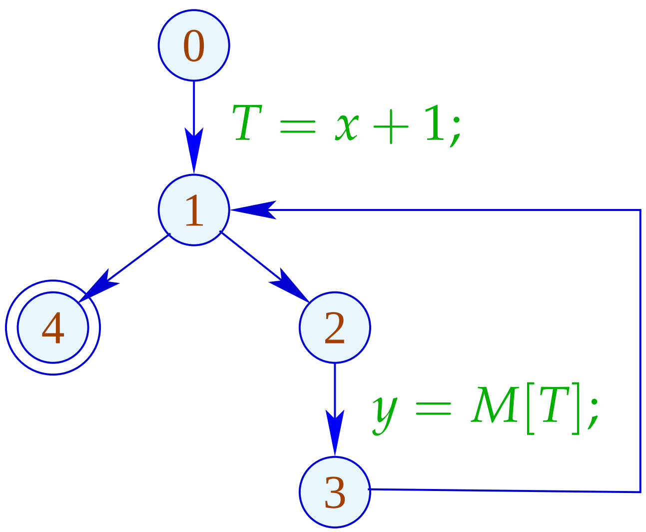
	$\mathcal{A}$
0	$\emptyset$
1	$\{x + 1\}$
2	$\{x + 1\}$
3	$\{x + 1\}$
4	$\{x + 1\}$

Fragen:

(1) Ist die Transformation korrekt ???

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

Beispiel:



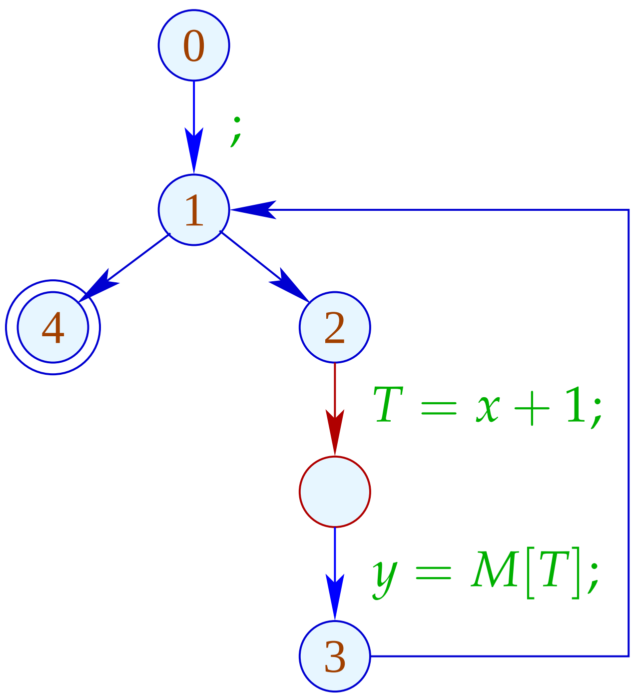
	$\mathcal{A}$
0	$\emptyset$
1	$\{x + 1\}$
2	$\{x + 1\}$
3	$\{x + 1\}$
4	$\{x + 1\}$

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:

→ Das Design einer sinnvollen Optimierung ist nicht ganz einfach.

→ Optimierungen, die ein Programm verbessern, können andere dramatisch verschlechtern :-(

→ 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:

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 :

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

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

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

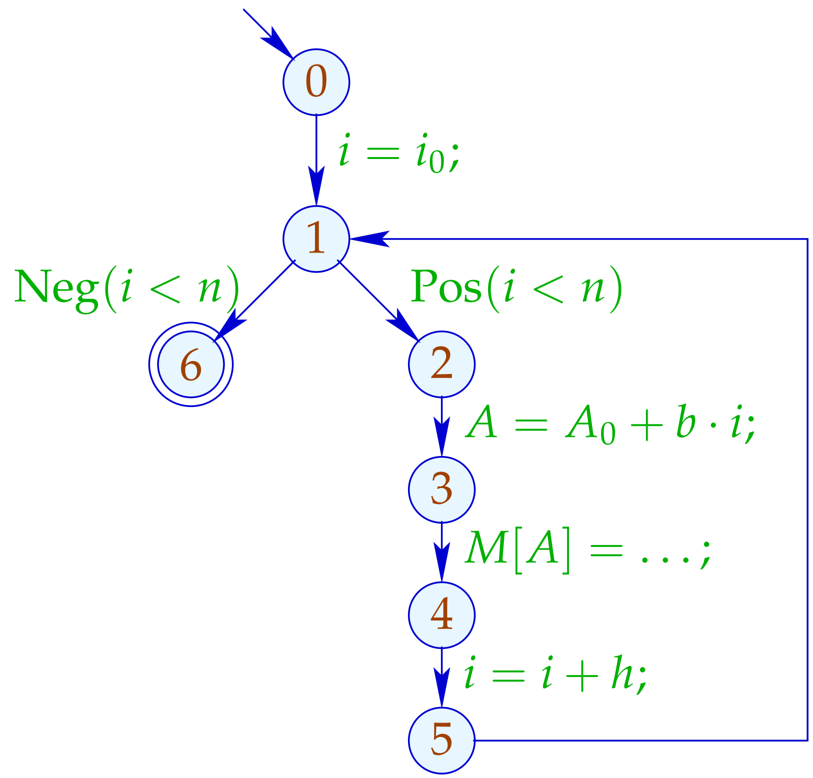
Beispiel:

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

$A = A_0 + b \cdot i;$

$M[A] = \ldots;$

}



... bzw. nach Schleifen-Rotation:

$i = i_0;$

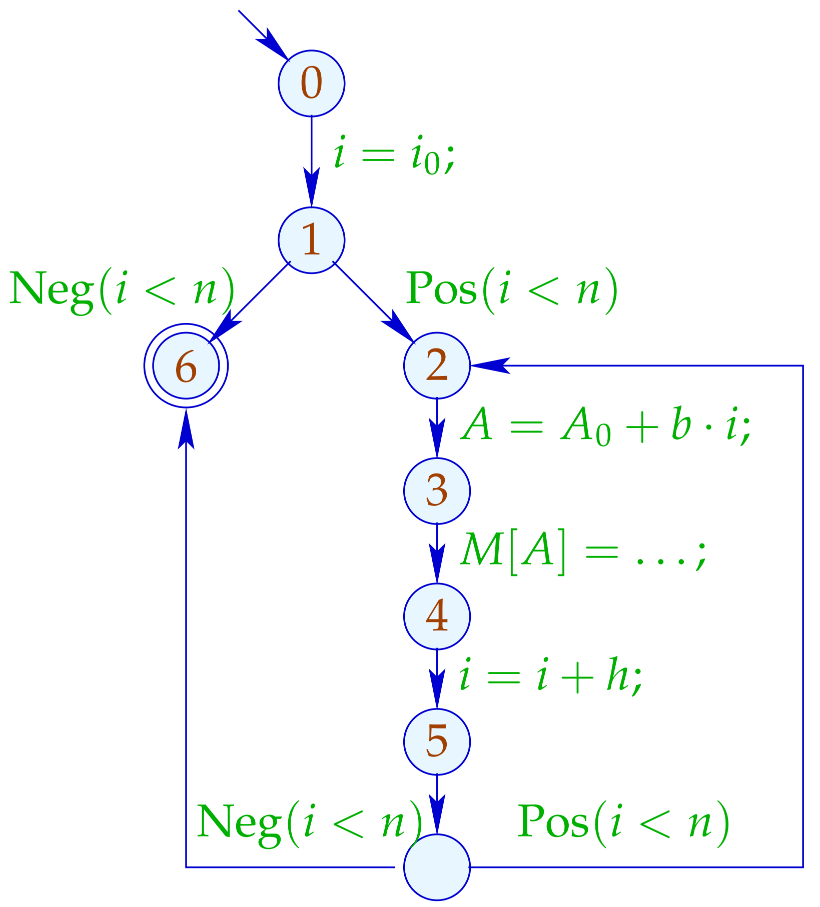
$\text{if } (i < n) \text{ do } \{$

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

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

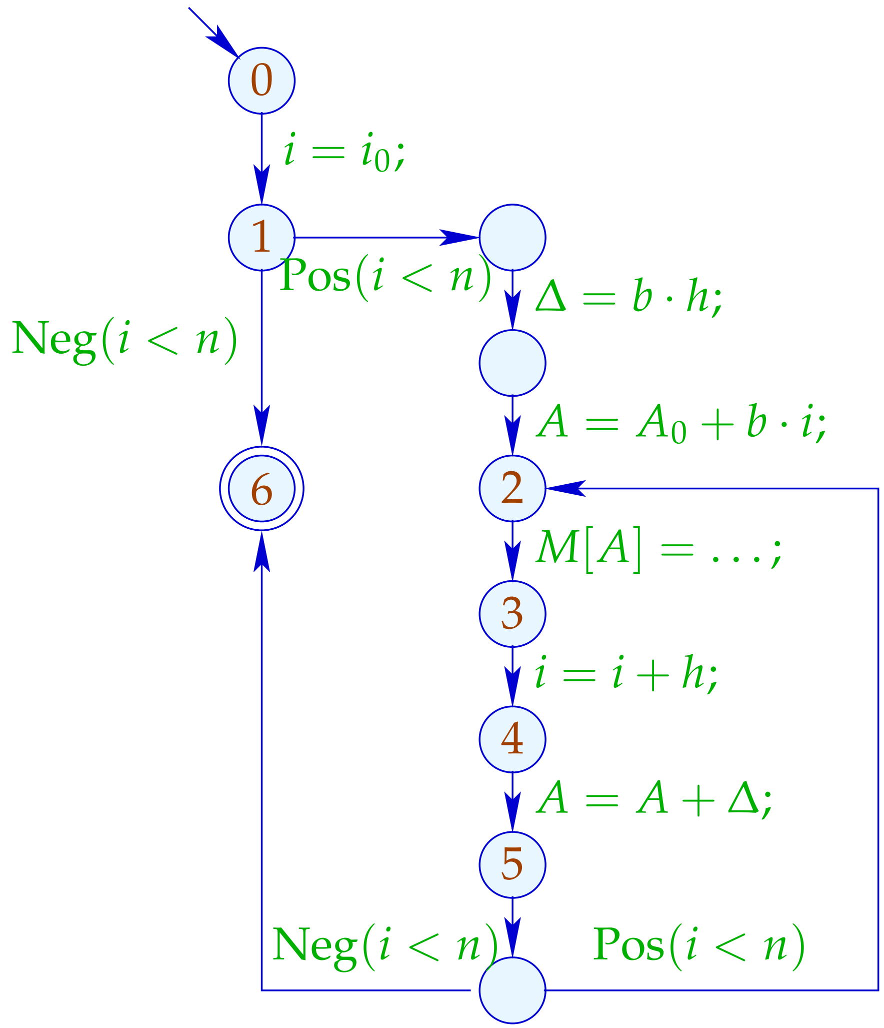
$\qquad i = i + h;$

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



... und Reduktion der Stärke:

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



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.

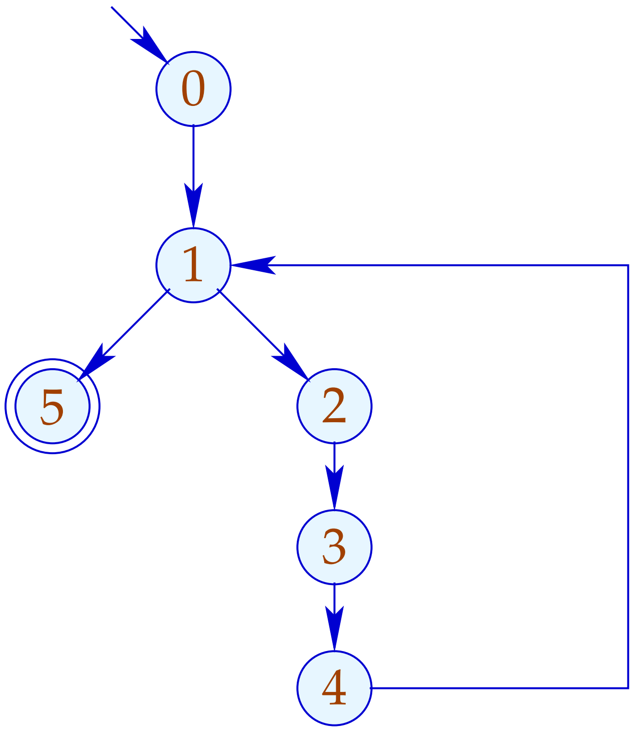
Schleifen:

... 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:

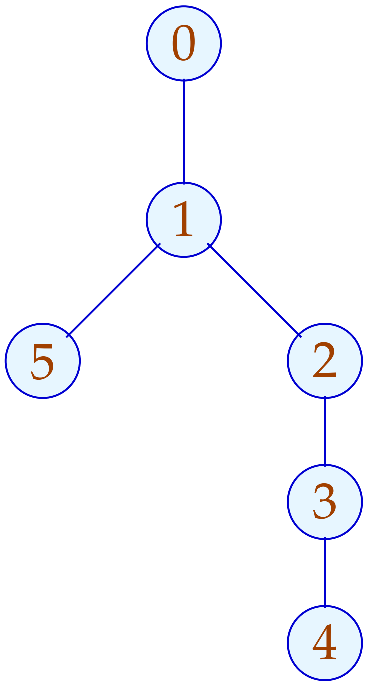
$$\mathsf{Loop}[v] \quad = \quad \{w \mid w \longrightarrow^* v \ \text{ in } \ G_v\}$$

Beispiel:



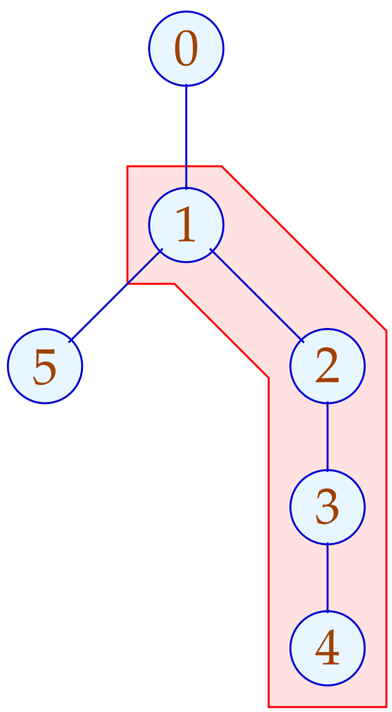
	$\mathcal{P}$
0	$\{0\}$
1	$\{0,1\}$
2	$\{0,1,2\}$
3	$\{0,1,2,3\}$
4	$\{0,1,2,3,4\}$
5	$\{0,1,5\}$

Beispiel:



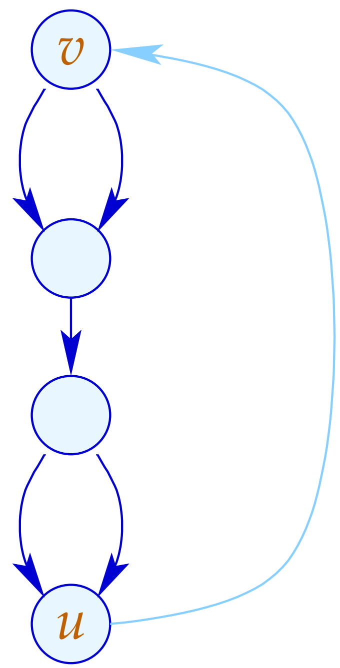
	$\mathcal{P}$
0	$\{0\}$
1	$\{0,1\}$
2	$\{0,1,2\}$
3	$\{0,1,2,3\}$
4	$\{0,1,2,3,4\}$
5	$\{0,1,5\}$

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:

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

Ist F distributiv, könnte man sie ersetzen durch:

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

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:

$$\begin{aligned} \Delta_{i+1} &= F(F^i(\emptyset)) \backslash F^i(\emptyset) \\ &= F(\Delta_i) \backslash (\Delta_1 \cup \ldots \cup \Delta_i) \quad \text{mit } \Delta_0 = \emptyset \end{aligned}$$

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 :-)

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

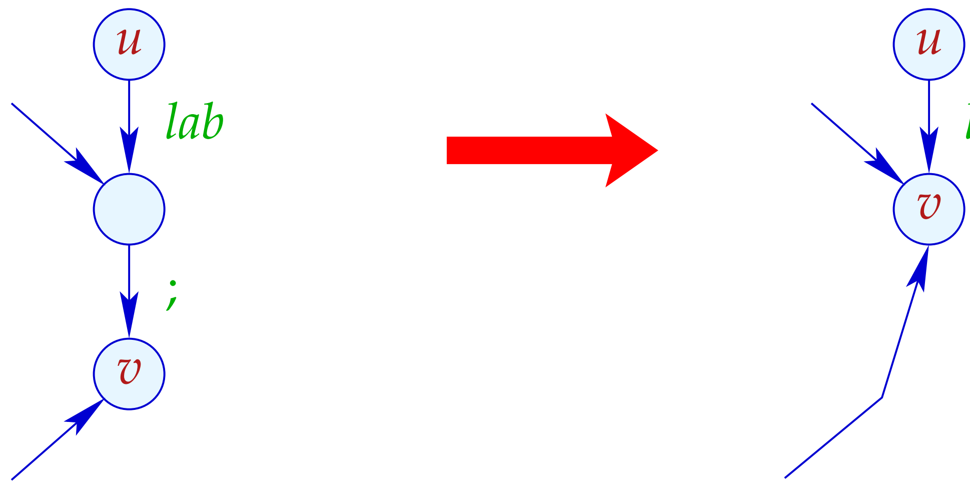
// algebraische Umformungen :-)

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

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

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

Wichtiges Teilproblem: *nop*-Optimierung



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

$\longrightarrow$ Folglich dürfen wir v_1 und v identifizieren :-)

$\longrightarrow$ 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 ???

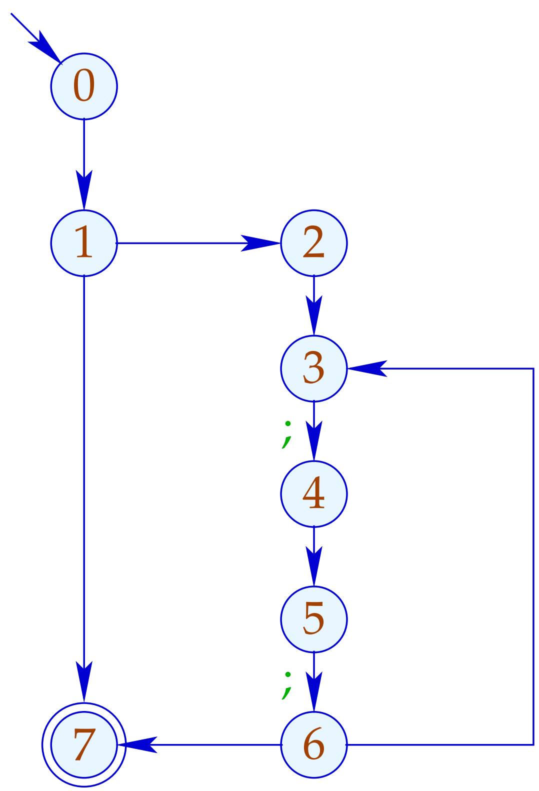
- Wir ersetzen jede Kante:

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

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

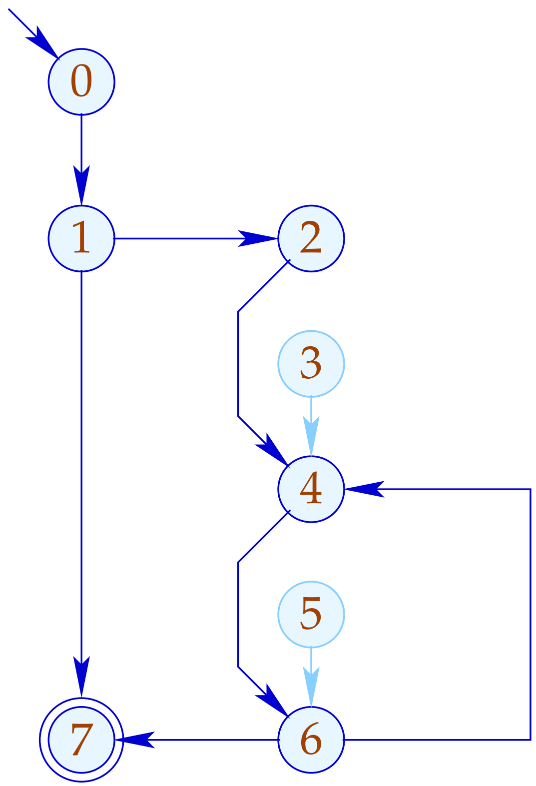
- Alle ;-Kanten werfen wir weg ;-)

Beispiel:



$$\begin{aligned} \text{next } 1 &= 1 \\ \text{next } 3 &= 4 \\ \text{next } 5 &= 6 \end{aligned}$$

Beispiel:



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

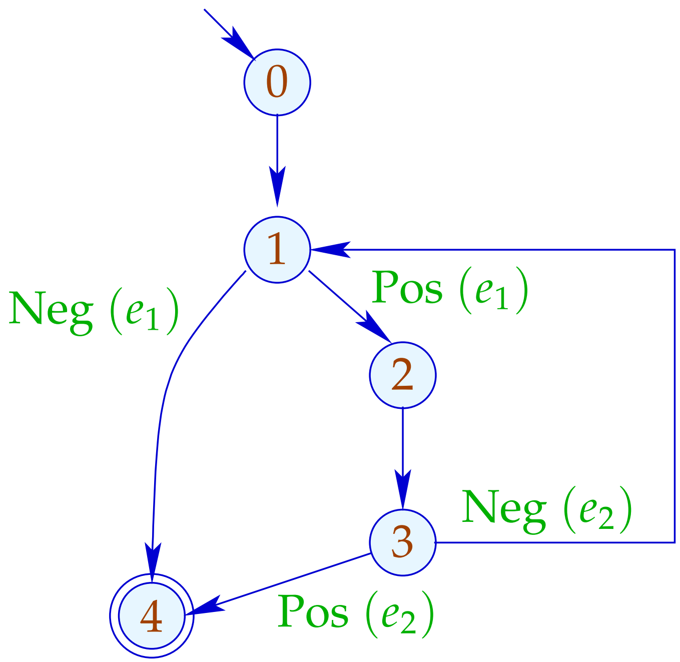
2. Teilproblem: Linearisierung

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

Achtung:

Nicht jede Linearisierung ist gleich gut !!!

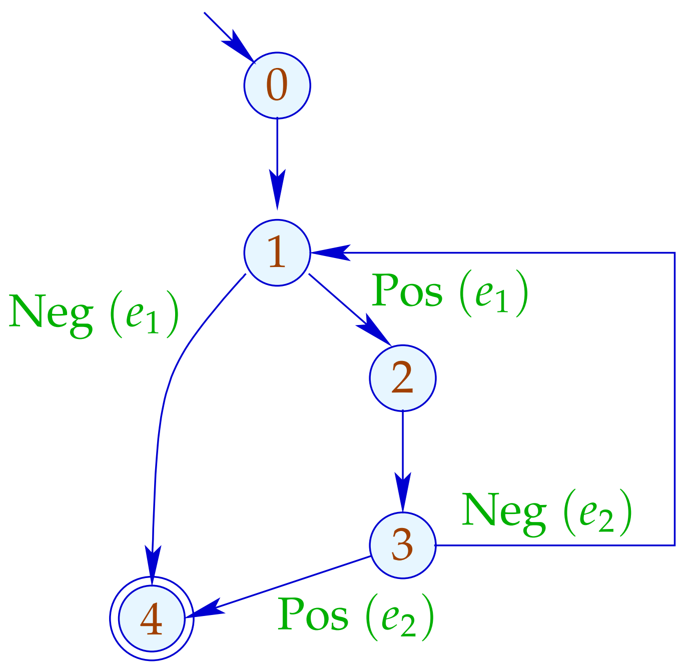
Beispiel:



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

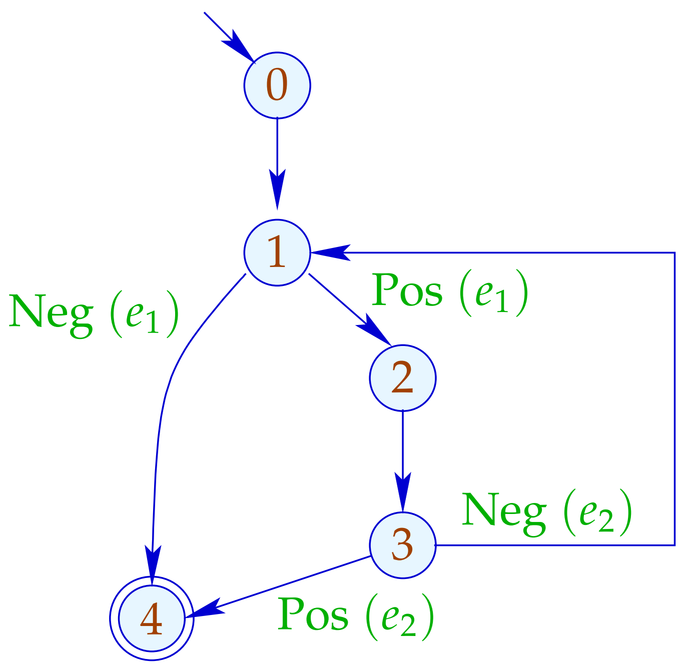
  (1) bereits behandelten Knoten;

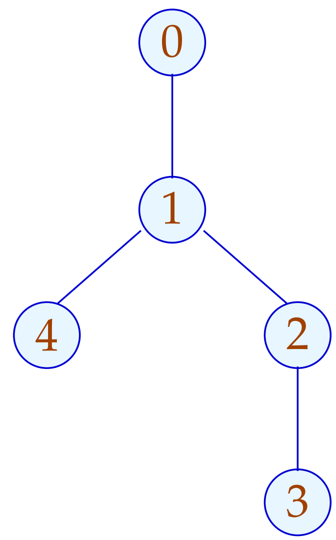
  (2) kälteren Knoten.

- Temperatur $\approx$ Schachtelungstiefe

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

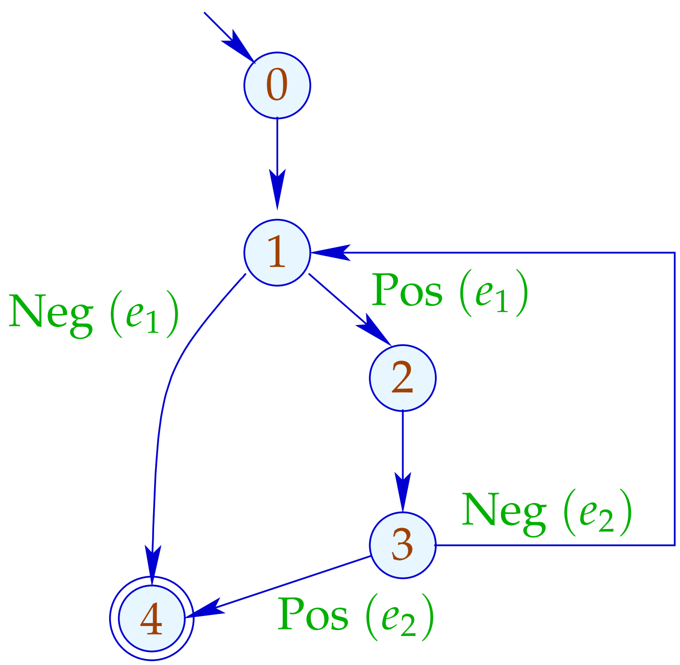
... im Beispiel:

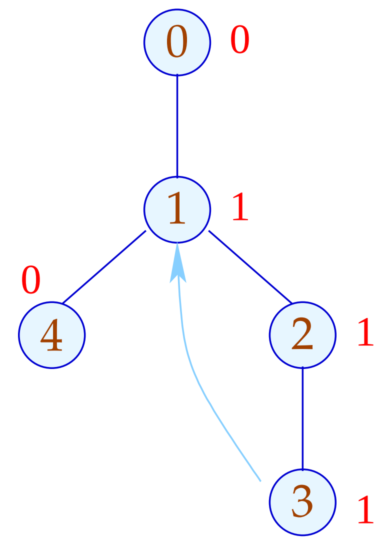




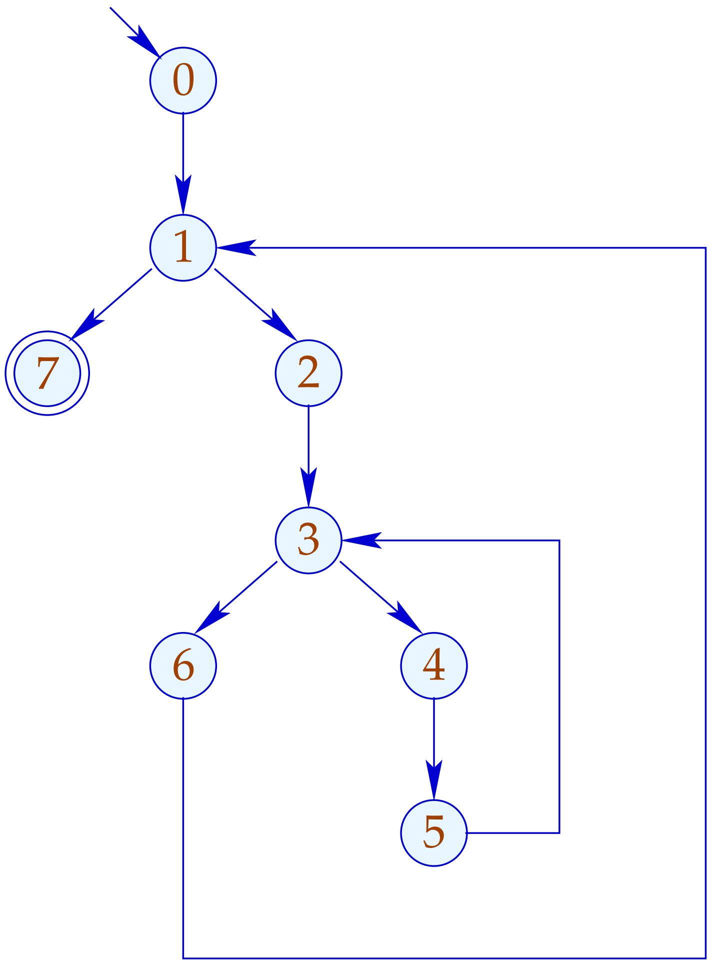
Der Teilbaum mit Rücksprung ist heißer ...

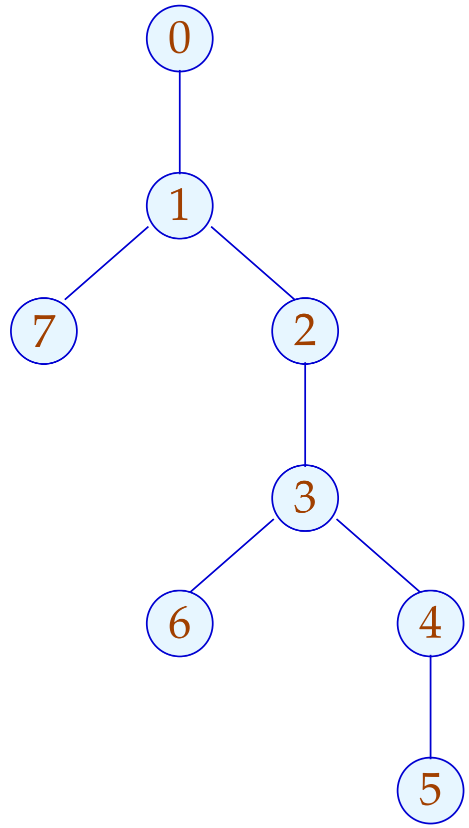
... im Beispiel:



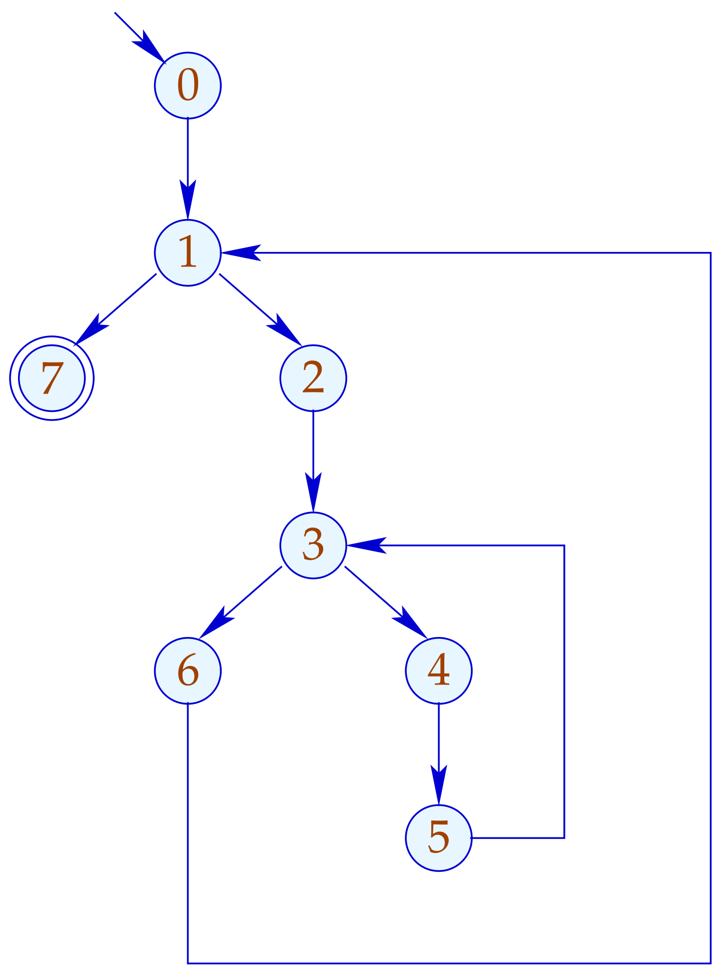


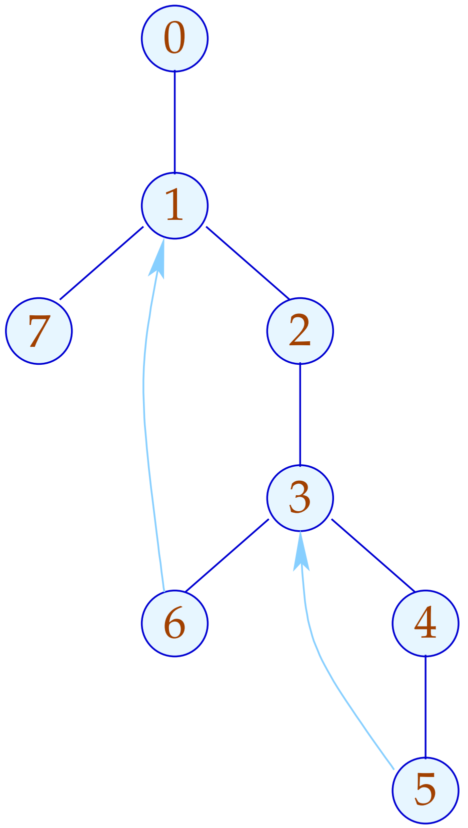
Komplizierteres Beispiel:



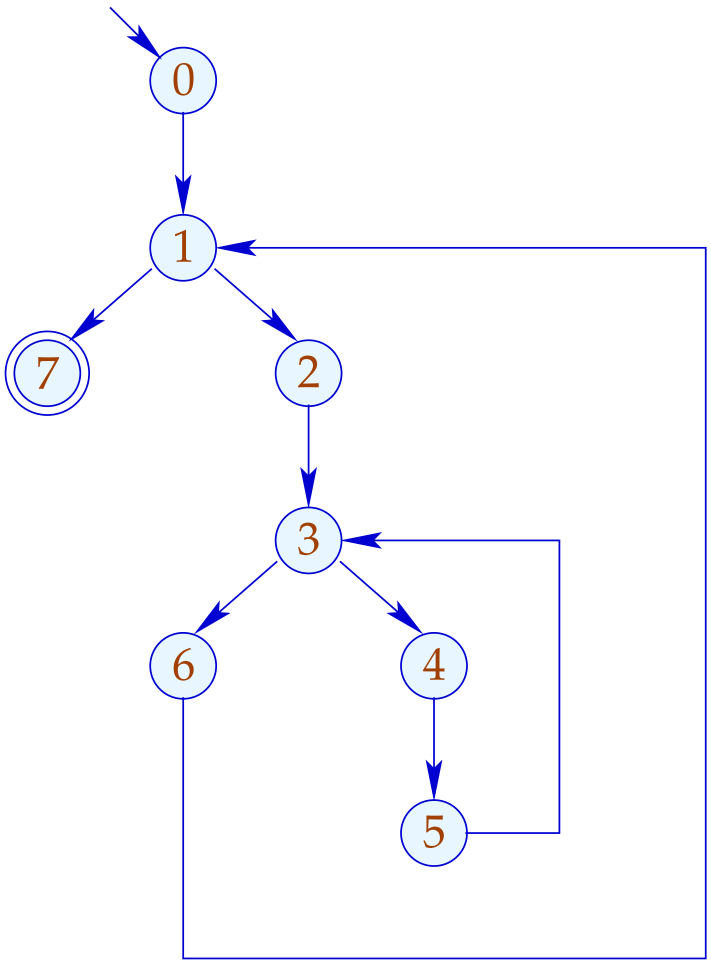


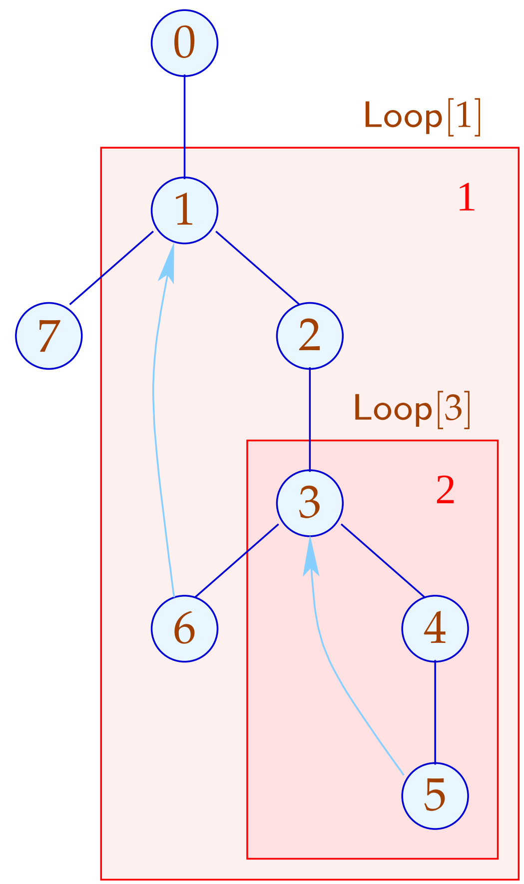
Komplizierteres Beispiel:





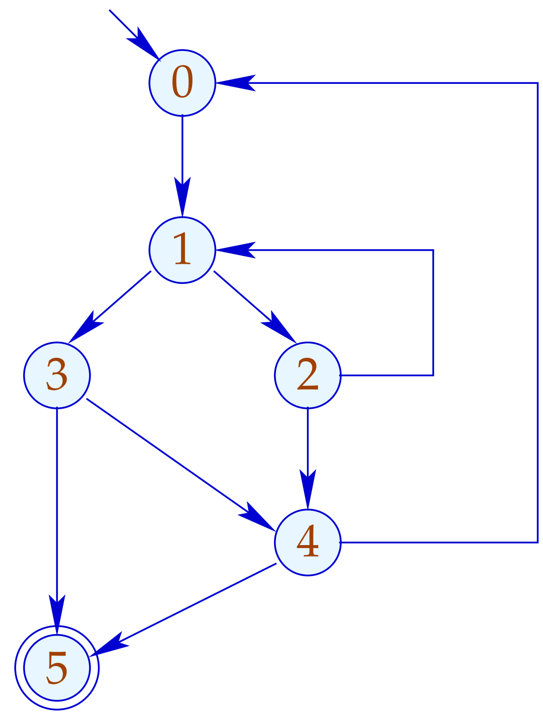
Komplizierteres Beispiel:

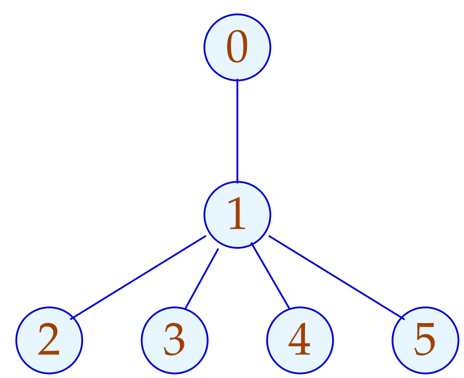




Unsere Definition von Loop sorgt dafür, dass (erkannte) Schleifen geschachtelt auftreten :-)

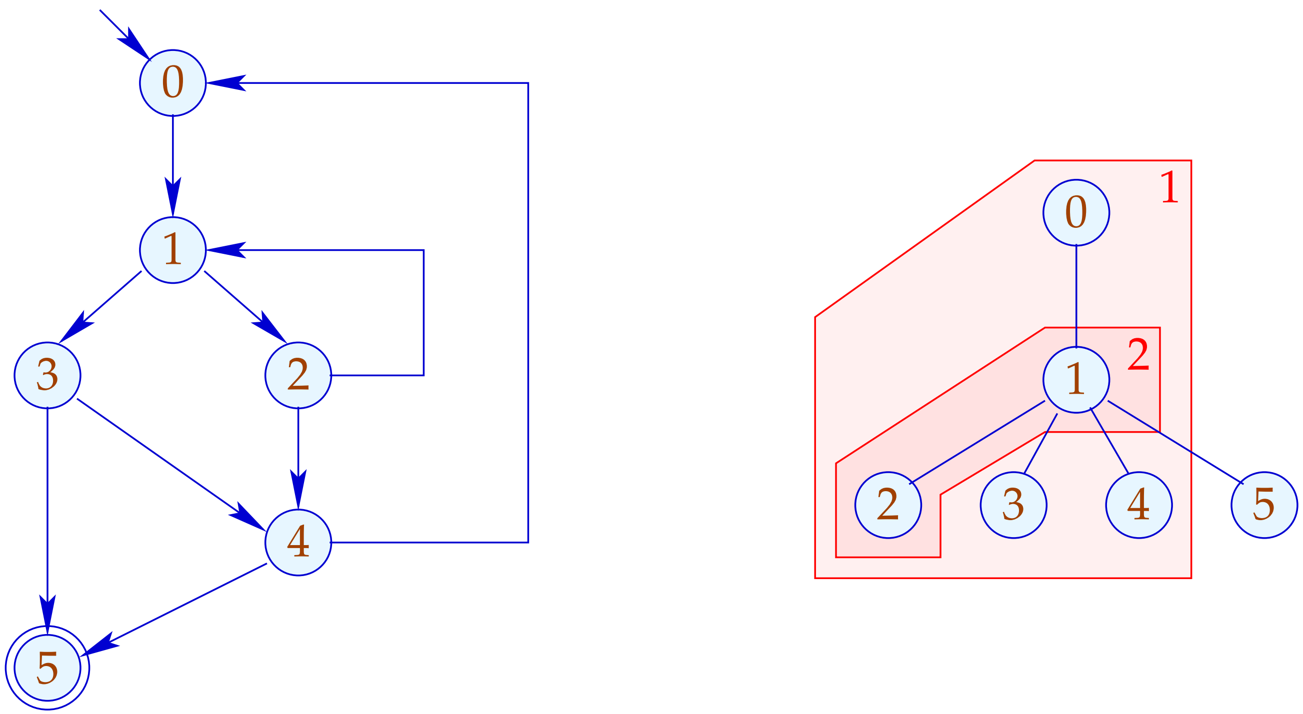
Sie ist auch für do-while-Schleifen mit breaks vernünftig...





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

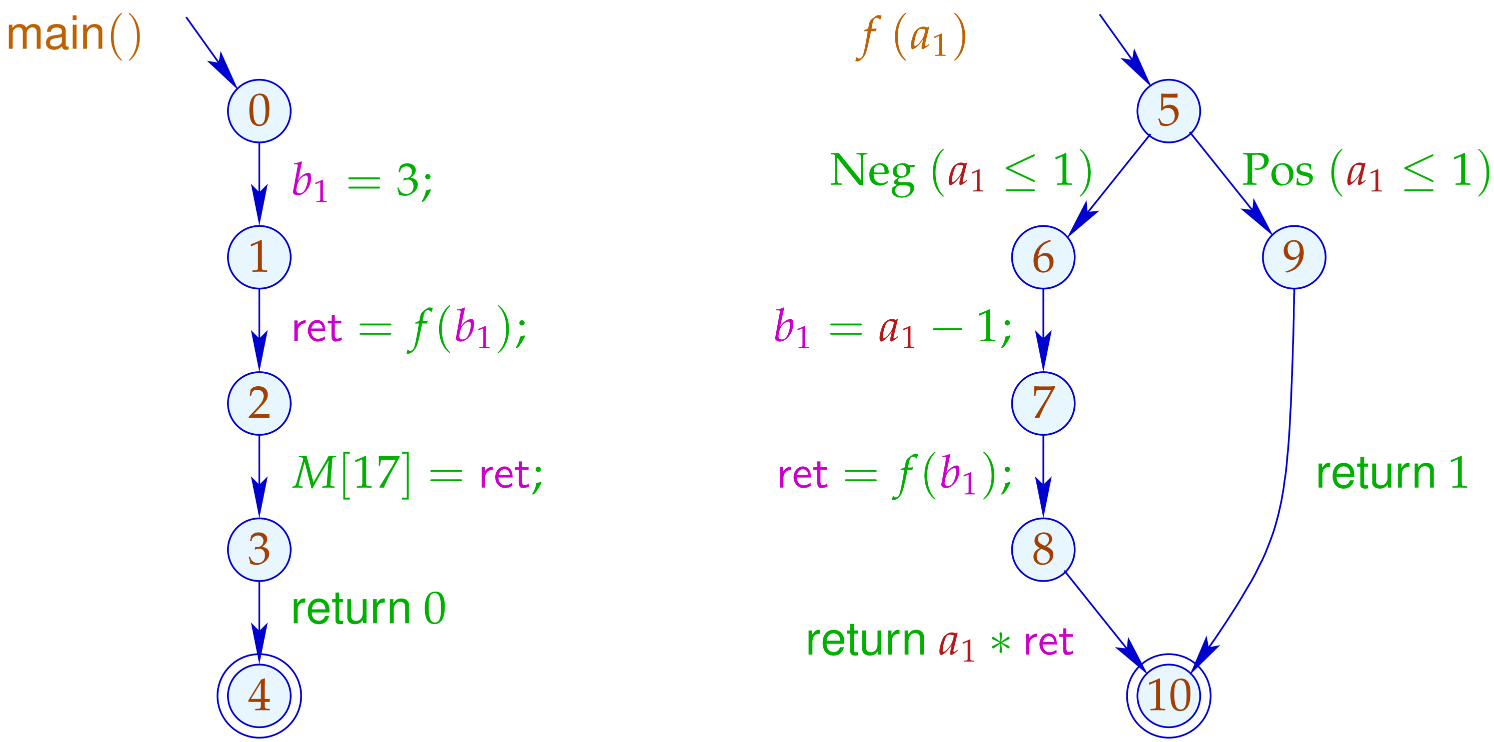
Die Programm-Ausführung startet mit dem Aufruf einer (parameterlosen) Funktion $\mathsf{main}\,()$.

Beispiel:

```
main () {                    f (a1) {
    b1 = 3;                      if (a1 ≤ 1) return 1;
    ret = f(b1);                    b1 = a1 − 1;
    M[17] = ret;                 ret = f(b1);
    return 0;                    return a1 · ret;
}                            }
```

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

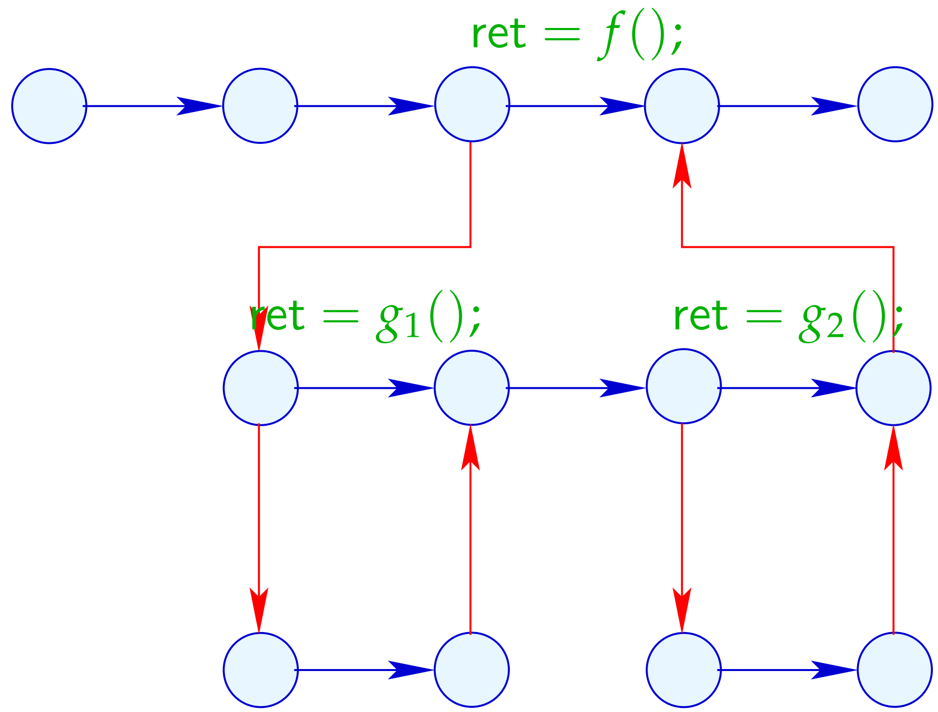
... im Beispiel:



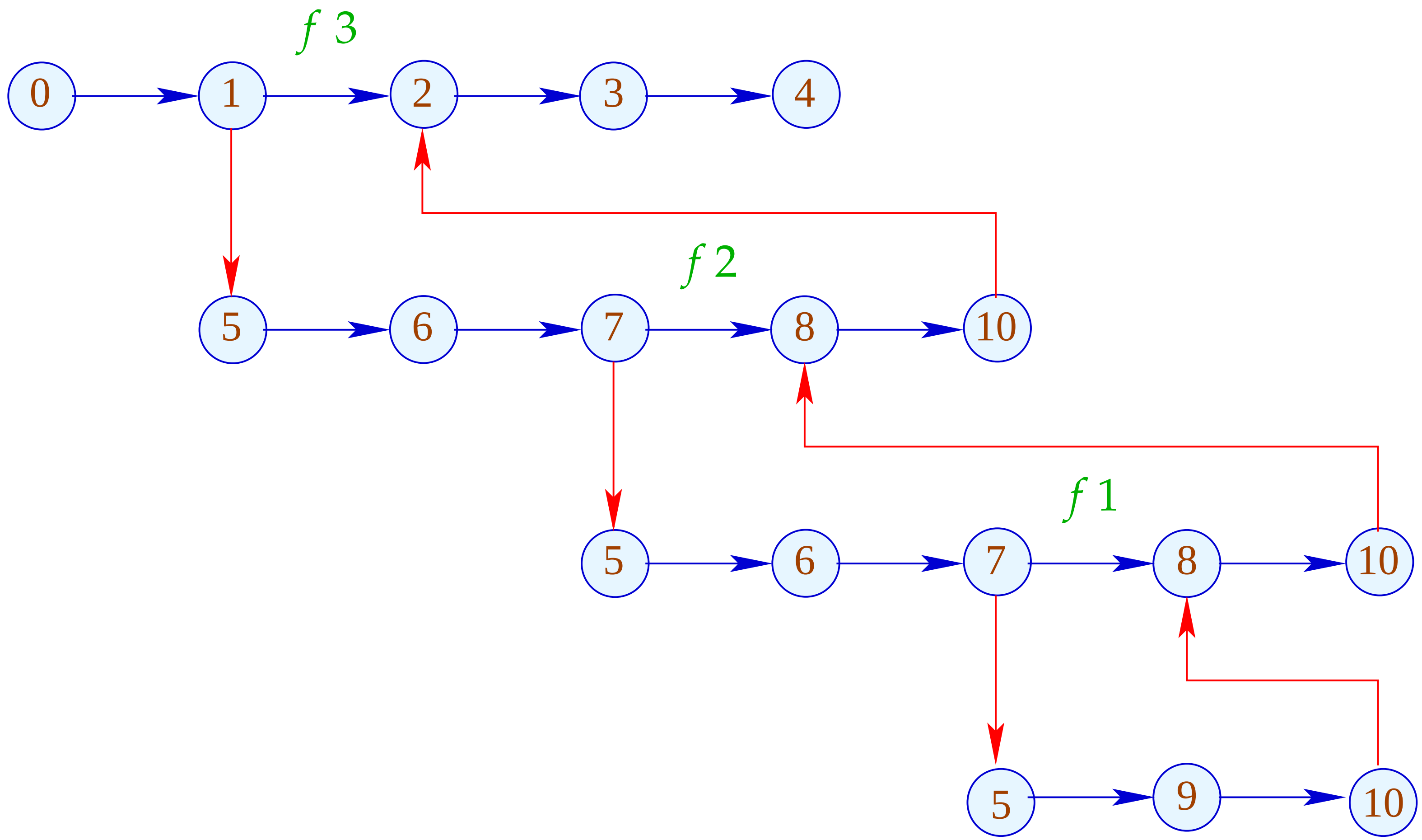
main()
0
$b_1 = 3;$
1
ret $= f(b_1);$
2
$M[17] =$ ret;
3
return 0
4
$f(a_1)$
5
Neg $(a_1 \leq 1)$
Pos $(a_1 \leq 1)$
6
9
$b_1 = a_1 - 1;$
7
ret $= f(b_1);$
return 1
8
return $a_1 *$ ret
10

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 :

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

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:

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

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

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

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

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

u_f Anfangspunkt von f

Rückkehr:

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

r_f Endpunkt von f

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

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

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

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

... im Beispiel:

10	$\mathsf{ret} \mapsto 2$
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	$\mathsf{ret} \mapsto 6$
2	

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

... im Beispiel:

2	ret $\mapsto 6$

Diese operationelle Semantik ist einigermaßen realistisch :-)

Kosten eines Prozedur-Aufrufs:

Vor Betreten des Rumpfs:
- Anlegen eines Kellerrahmens;
- Retten der Register;
- Retten der Fortsetzungsaddresse;
- Anspringen des Rumpfs.

Bei Beenden des Aufrufs:
- Aufgeben des Kellerrahmens;
- Restaurieren der Register;
- Übergeben des Ergebnisses;
- Rücksprung hinter die Aufrufstelle.

$\Longrightarrow$ ... ziemlich teuer !!!

1. Idee: Inlining

Kopiere den Funktionsrumpf an jede Aufrufstelle !!!

Beispiel:

$$
\begin{array}{ll}
abs\ (a_1)\ \{ & max\ (a_1, a_2)\ \{ \\
\quad b_1 = a_1; & \quad \text{if}\ (a_1 < a_2)\ \text{return}\ a_2; \\
\quad b_2 = -a_1; & \quad \text{return}\ a_1; \\
\quad \text{ret} = max\ (b_1, b_2); & \} \\
\quad \text{return}\ \text{ret}; & \\
\} &
\end{array}
$$

... liefert:

$$
\begin{array}{ll}
abs\ (a_1)\ \ \{ & \\
& b_1 = a_1; \\
& b_2 = -a_1; \\
& a_1 = b_1; \\
& a_2 = b_2; \\
& \boxed{\begin{array}{l} \text{if}\ \ (a_1 < a_2)\ \ \{\ \ \text{ret} = a_2;\ \ \text{goto}\ \ \_max;\ \ \} \\ \text{ret} = a_1;\ \ \text{goto}\ \ \_max; \end{array}} \\
\_max: & \text{return}\ \ \text{ret}; \\
\} &
\end{array}
$$

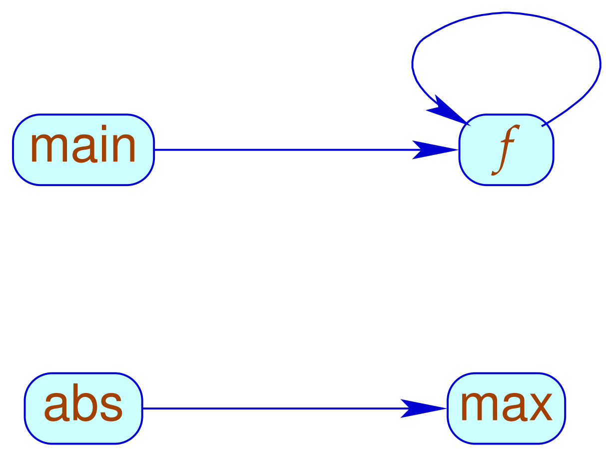
Probleme:

- Der einkopierte Block modifiziert evt. die a_i, b_i :-(
- Allgemeiner: Mehrfachbenutzung gleicher Variablennamen kann zu Fehlern führen;
- Mehrfach-Verwendung einer Funktion führt zu Code-Duplizierung :-((
- Wie gehen wir mit Rekursion um ???

Erkennen von Rekursion:

Wir konstruieren den Aufruf-Graph des Programms.

In den Beispielen:



Aufruf-Graph:

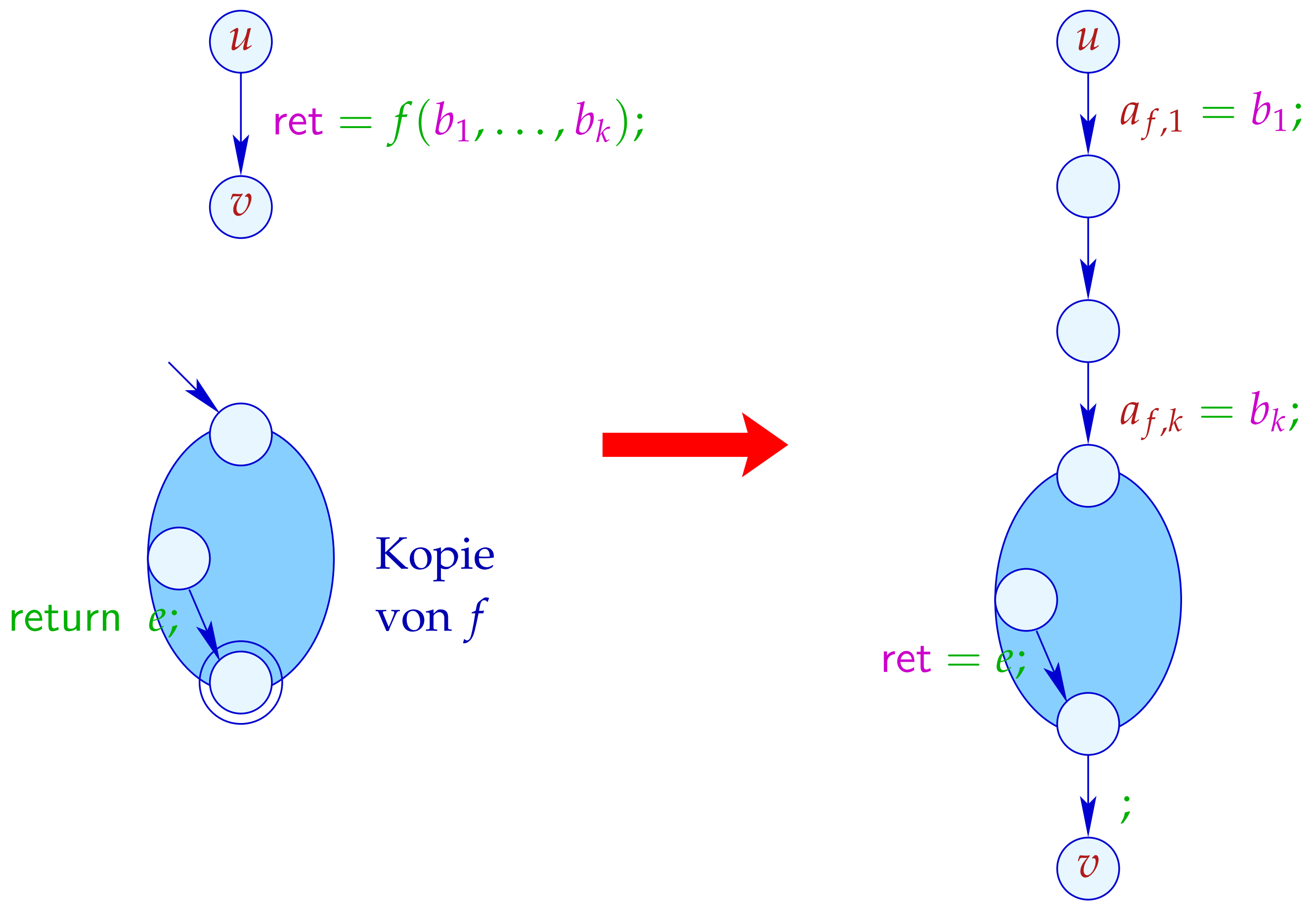
- Die Knoten sind die Funktionen.
- Eine Kante geht von g nach h, sofern der Rumpf von g einen Aufruf von h enthält.

Strategien für Inlining:

- Kopiere nur Blatt-Funktionen ein, d.h. solche ohne weitere Aufrufe :-)
- Kopiere sämtliche nicht-rekursiven Funktionen ein!

... wir betrachten hier nur Blatt-Funktionen ;-)

Transformation 10:



Beachte:

- Blatt-Funktionen benutzen keine Variablen b_i :-)
- Die Variable ret der ein-kopierten Funktion wird mit der Variablen ret der aufrufenden identifiziert.

  Das spart uns eine Umspeicherung :-)
- Die Nop-Kante können wir ebenfalls einsparen, da der *stop*-Knoten von f selbst keine ausgehenden Kanten hat ...
- Die $a_{f,i}$ sind die Kopien der aktuellen Parameter von f.
- Eigentlich müssten wir alle anderen Variablen von f noch mit 0 initialisieren :-)

2. Idee: Beseitigung von Endrekursion

Betrachte das folgende Beispiel:

$$
\begin{array}{l}
f\,(a_1, a_2)\ \{ \\
\qquad \mathsf{if}\ \ (a_2 \leq 1)\ \ \mathsf{return}\ \ a_1; \\
\qquad b_1 = a_1 \cdot a_2; \\
\qquad b_2 = a_1 - 1; \\
\qquad \mathsf{ret} = f\,(b_1, b_2); \\
\qquad \mathsf{return}\ \ \mathsf{ret}; \\
\}
\end{array}
$$

Der Funktionsaufruf im Rumpf liefert das Ergebnis.

Nach dem Funktionsaufruf gibt es im Rumpf nichts mehr zu tun.

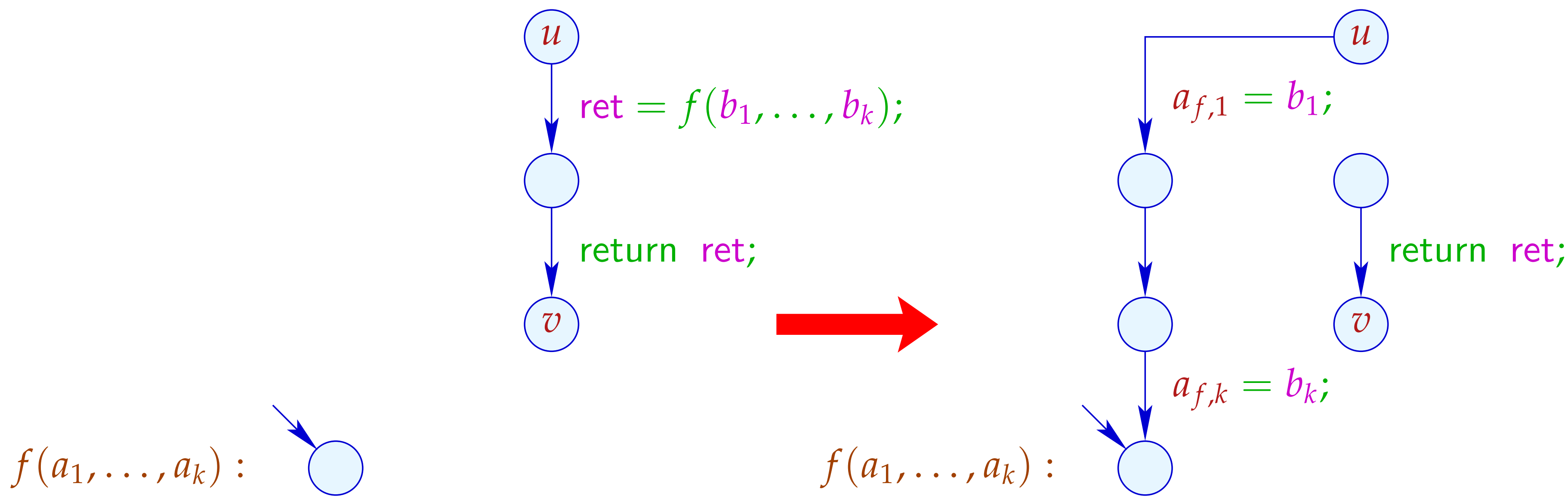
$\Longrightarrow$ Wir könnten ihn direkt anspringen :-)

... das liefert im Beispiel:

$$
\begin{array}{l}
f\,(a_1, a_2)\ \ \{ \\
\quad \_f:\ \ \mathsf{if}\ \ (a_2 \leq 1)\ \ \mathsf{return}\ \ a_1; \\
\qquad\quad b_1 = a_1 \cdot a_2; \\
\qquad\quad b_2 = a_1 - 1; \\
\qquad\quad a_1 = b_1; \\
\qquad\quad a_2 = b_2; \\
\qquad\quad \mathsf{goto}\ \ \_f; \\
\}
\end{array}
$$

// Das funktioniert, weil wir Referenzen auf Variablen
// verbieten.

Transformation 11:



Achtung:

→ Diese Optimierung ist besonders wichtig bei Programmiersprachen ohne Iterationskonstrukte !!!

→ Duplizieren von Code ist nicht erforderlich :-)

→ Variablen brauchen nicht umbenannt zu werden :-)

→ Die Optimierung hilft auch bei nicht-rekursiven Endaufrufen :-)

→ Der entstehende Code kann Sprünge aus dem Rumpf einer Funktion in eine andere enthalten ???

Exkurs 4: Interprozedurale Analyse

Bisher können wir nur jede Funktion einzeln analysieren.

→ Die Kosten halten sich in Grenzen :-)

→ Die Techniken funktionieren auch bei getrennter Übersetzung :-)

→ An Funktionsaufrufen müssen wir das Schlimmste annehmen :-(

→ Konstantenpropagation funktioniert nur für lokale Konstanten :-((

Frage:

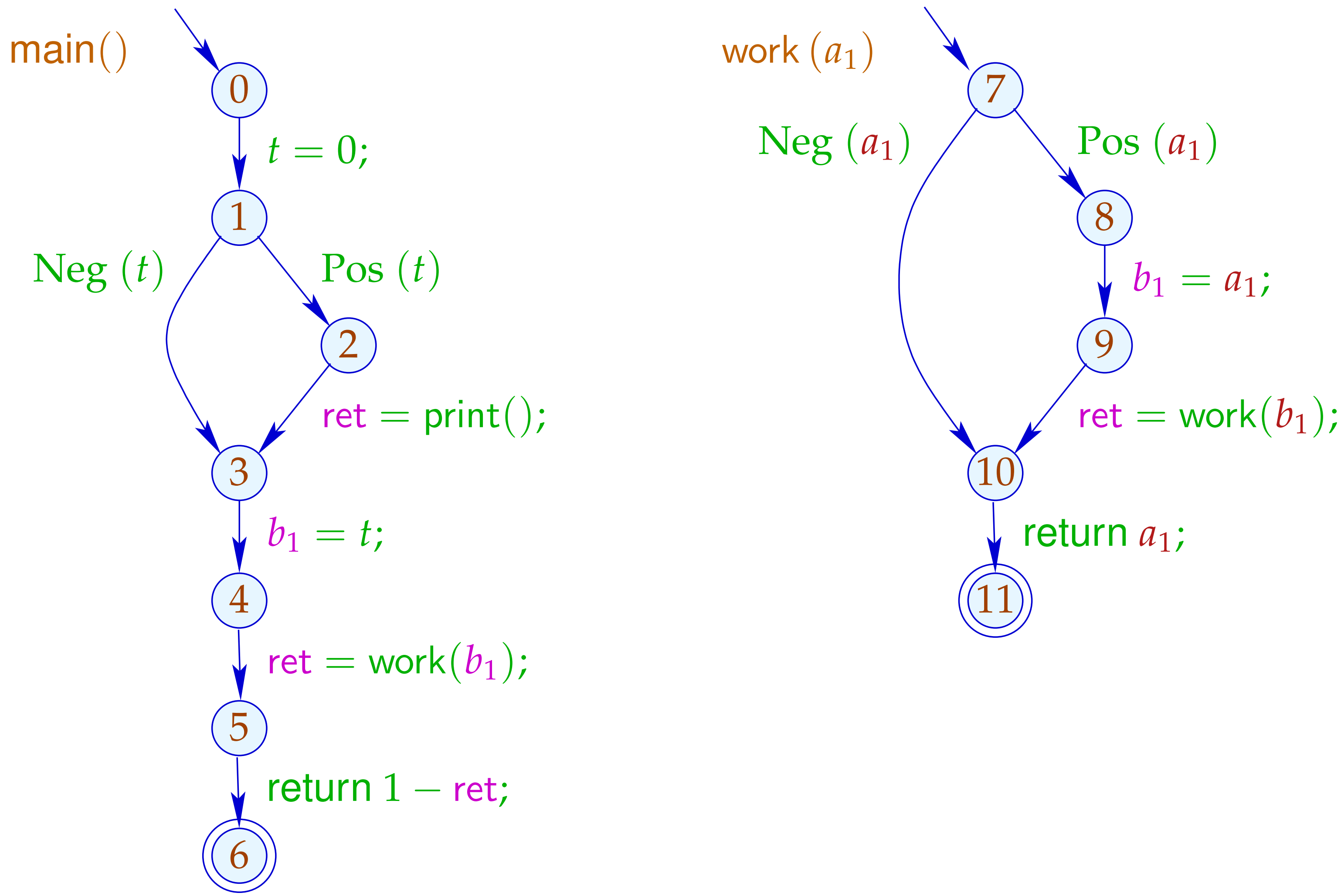
Wie analysiert man rekursive Programme ???

Beispiel: Konstantenpropagation

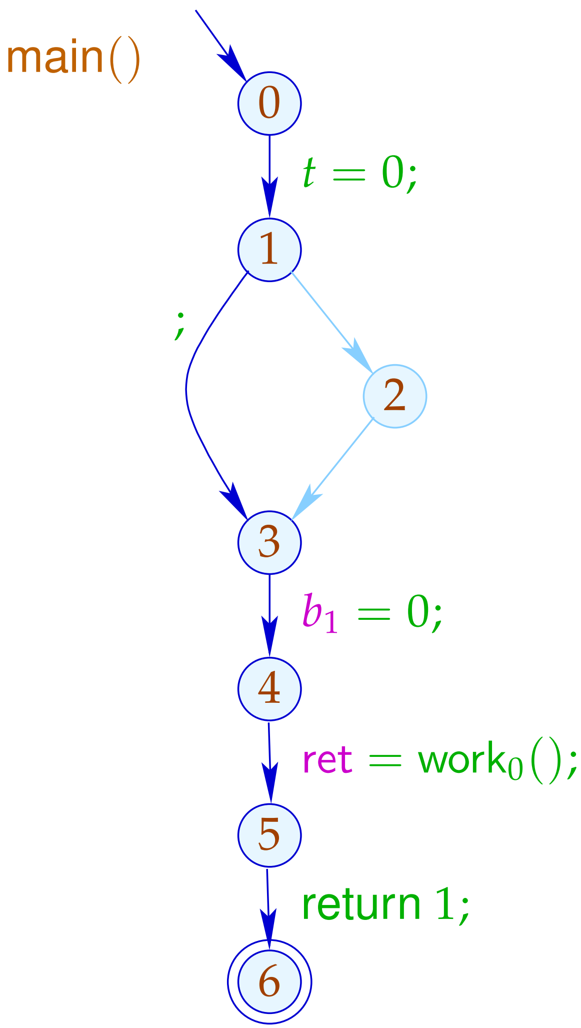
```
main() {
    t = 0;
    if (t) print();
    b1 = t;
    ret = work (b1);
    return 1 − ret;
}

work(a1) {
    if (a1) {
        b1 = a1;
        work(b1);
    }
    return a1;
}
```

Beispiel: Konstantenpropagation



Beispiel: Konstantenpropagation



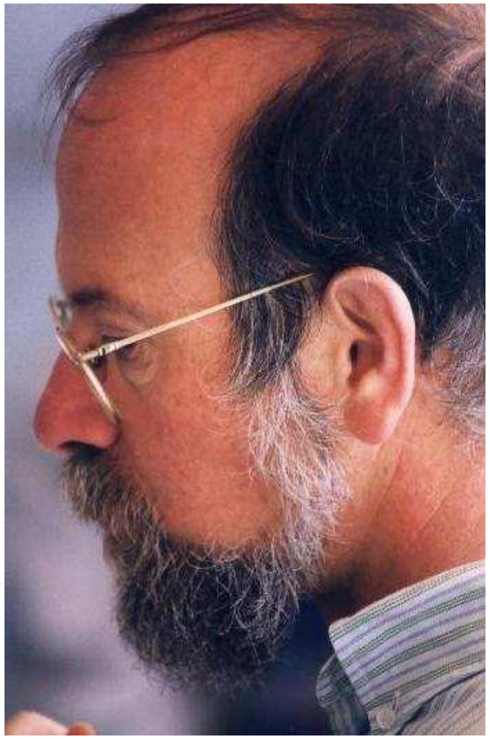
(1) Funktionaler Ansatz:

Sei $\mathbb{D}$ ein vollständiger Verband von (abstrakten) Zuständen.

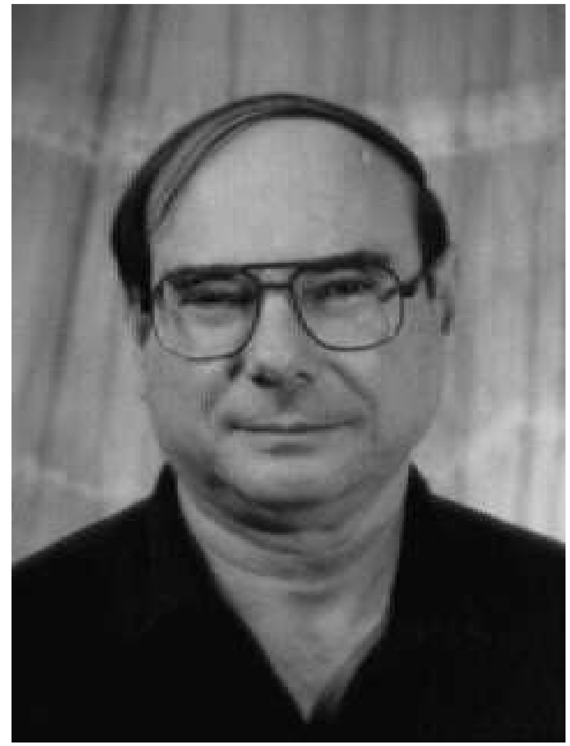
Idee:

Repräsentiere den Effekt von $f()$ durch eine Funktion:

$$[\![f]\!]^\sharp \;:\; \mathbb{D} \to \mathbb{D}$$



Micha Sharir, Tel Aviv University



Amir Pnueli, Weizmann Institute

Um den Effekt einer Aufrufskante $k = (u, \mathsf{ret} = f\,(b_1, \ldots, b_k);, v)$ zu ermitteln, benötigen wir abstrakte Funktionen:

$$
\begin{array}{lll}
\mathsf{enter}^\sharp_f & : & \mathbb{D} \longrightarrow \mathbb{D} \\
\mathsf{combine}^\sharp & : & \mathbb{D}^2 \longrightarrow \mathbb{D}
\end{array}
$$

Damit erhalten wir:

$$
[\![k]\!]^\sharp\, D \quad = \quad \mathsf{combine}^\sharp\,(D, [\![f]\!]^\sharp\,(\mathsf{enter}^\sharp_f\, D))
$$

... für Konstantenpropagation:

$$\mathbb{D} = (\mathit{Vars} \to \mathbb{Z}^\top)_\bot$$

$$\mathsf{enter}_f^\sharp\; D = \begin{cases} \bot & \text{falls} \quad D = \bot \\ \{a_i \mapsto (D\; b_i) \mid i = 1, \ldots, k\} & \text{sonst} \end{cases}$$

$$\mathsf{combine}^\sharp\; (D_1, D_2) = \begin{cases} \bot & \text{falls} \quad D_1 = \bot \vee D_2 = \bot \\ D_1 \oplus \{\mathsf{ret} \mapsto (D_2\; \mathsf{ret})\} & \text{sonst} \end{cases}$$

Um die Effekte $[\![f]\!]^\sharp$ zu ermitteln, stellen wir ein Constraint-System über dem vollständigen Verband $\mathbb{D} \to \mathbb{D}$ auf:

$$\begin{array}{llll} [\![v]\!]^\sharp & \sqsupseteq & \mathsf{Id} & v \quad \text{Eintrittspunkt} \\ [\![v]\!]^\sharp & \sqsupseteq & [\![k]\!]^\sharp \circ [\![u]\!]^\sharp & k = (u, \_, v) \quad \text{Kante} \\ [\![f]\!]^\sharp & \sqsupseteq & [\![stop_f]\!]^\sharp & stop_f \quad \text{Endpunkt von } f \end{array}$$

$[\![v]\!]^\sharp \; : \; \mathbb{D} \to \mathbb{D}$ beschreibt den Effekt aller Präfixe der Berechnungswälder w einer Funktion, die vom Eintrittspunkt nach v führen :-)

Probleme:

- Wie beschreibt man eine Funktion $f \;:\; \mathbb{D} \to \mathbb{D}$???
- Ist $\#\mathbb{D} = \infty$, hat $\mathbb{D} \to \mathbb{D}$ unendliche aufsteigende Ketten :-(

Vereinfachung: Kopier-Konstanten

$\longrightarrow$ Bedingungen interpretieren wir wie ein $;$:-)

$\longrightarrow$ Wir behandeln exakt nur Zuweisungen $x = e;$ mit $e \in \mathit{Vars} \cup \mathbb{Z}$:-)

Beobachtung:

→ Die Effekte von Zuweisungen sind:

$$[\![x = e;]\!]^\sharp\, D \quad = \quad \begin{cases} D \oplus \{x \mapsto c\} & \text{falls} \quad e = c \in \mathbb{Z} \\ D \oplus \{x \mapsto (D\; y)\} & \text{falls} \quad e = y \in \mathit{Vars} \\ D \oplus \{x \mapsto \top\} & \text{sonst} \end{cases}$$

→ Sei $\mathbb{V}$ die (endliche !!!) Menge der konstanten rechten Seiten. Dann haben Variablen stets Werte aus $\mathbb{V}^\top$:-))

→ Die auftretenden Effekte sind enthalten in

$$\mathbb{D}_f \to \mathbb{D}_f \qquad \text{mit} \qquad \mathbb{D}_f = (\mathit{Vars} \to \mathbb{V}^\top)_\perp$$

→ Dieser Verband ist riesig, aber endlich !!!

Verbesserung:

→ Nicht alle Funktionen aus $\mathbb{D}_f \to \mathbb{D}_f$ kommen wirklich vor :-)

→ Alle vorkommenden Funktionen $\lambda D.\bot \neq M$ sind von der Form:

$$M \quad = \quad \{x \mapsto (b_x \sqcup \bigsqcup_{y \in I_x} y) \mid x \in Vars\} \qquad \text{wobei:}$$
$$M\;D \quad = \quad \{x \mapsto (b_x \sqcup \bigsqcup_{y \in I_x} D\;y) \mid x \in Vars\} \qquad \text{für} \quad D \neq \bot$$

→ Sei $\mathbb{M}$ die Menge aller dieser Funktionen. Dann gilt für $M_1, M_2 \in \mathbb{M}$ $(M_1 \neq \lambda\,D.\,\bot \neq M_2)$:

$$(M_1 \sqcup M_2)\;x \quad = \quad (M_1\;x) \sqcup (M_2\;x)$$

→ Für $k = \#Vars$ hat $\mathbb{M}$ die Höhe $(k+1) \cdot k + 1$:-)

Verbesserung (Forts.):

$\longrightarrow$ Auch die Komposition lässt sich direkt implementieren:

$$
\begin{array}{rcll}
(M_1 \circ M_2)\ x & = & b' \sqcup \bigsqcup_{y \in I'} y & \quad \text{mit} \\
b' & = & b \sqcup \bigsqcup_{z \in I} b_z & \\
I' & = & \bigcup_{z \in I} I_z & \quad \text{sofern} \\
M_1\ x & = & b \sqcup \bigsqcup_{y \in I} y & \\
M_2\ z & = & b_z \sqcup \bigsqcup_{y \in I_z} y &
\end{array}
$$

$\longrightarrow$ Die Effekte von Zuweisungen sehen dann so aus:

$$
[\![x = e;]\!]^\sharp = \begin{cases}
\mathsf{Id}_{Vars} \oplus \{x \mapsto c\} & \text{falls} \quad e = c \in \mathbb{Z} \\
\mathsf{Id}_{Vars} \oplus \{x \mapsto y\} & \text{falls} \quad e = y \in Vars \\
\mathsf{Id}_{Vars} \oplus \{x \mapsto \top\} & \text{sonst}
\end{cases}
$$

... im Beispiel:

$$
\begin{array}{lll}
[\![t = 0;]\!]^\sharp & = & \{a_1 \mapsto a_1, b_1 \mapsto b_1, \mathsf{ret} \mapsto \mathsf{ret}, \boxed{t \mapsto 0}\} \\
[\![b_1 = a_1;]\!]^\sharp & = & \{a_1 \mapsto a_1, \boxed{b_1 \mapsto a_1}, \mathsf{ret} \mapsto \mathsf{ret}, t \mapsto t\}
\end{array}
$$

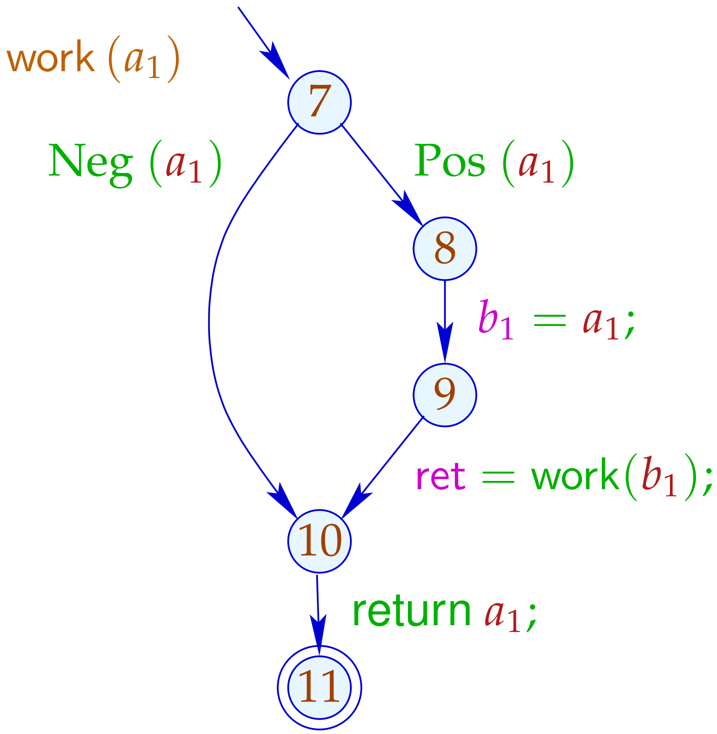
Um die Analyse zu implementieren, müssen wir nur noch den Effekt eines Aufrufs $k = (\_, \mathsf{ret} = f\,(b_1, \ldots, b_k);, \_)$ aus dem Effekt der Funktion f ermitteln:

$$
\begin{array}{lll}
[\![k]\!]^\sharp & = & H_f\,([\![f]\!]^\sharp) \qquad \text{wobei:} \\
H_f\,(M) & = & \mathsf{Id} \oplus \{\mathsf{ret} \mapsto ((M \circ E_f)\ \mathsf{ret})\} \\
E_f\ x & = & \begin{cases} b_i & \text{falls} \quad x = a_i \\ 0 & \text{sonst} \end{cases}
\end{array}
$$

... im Beispiel:

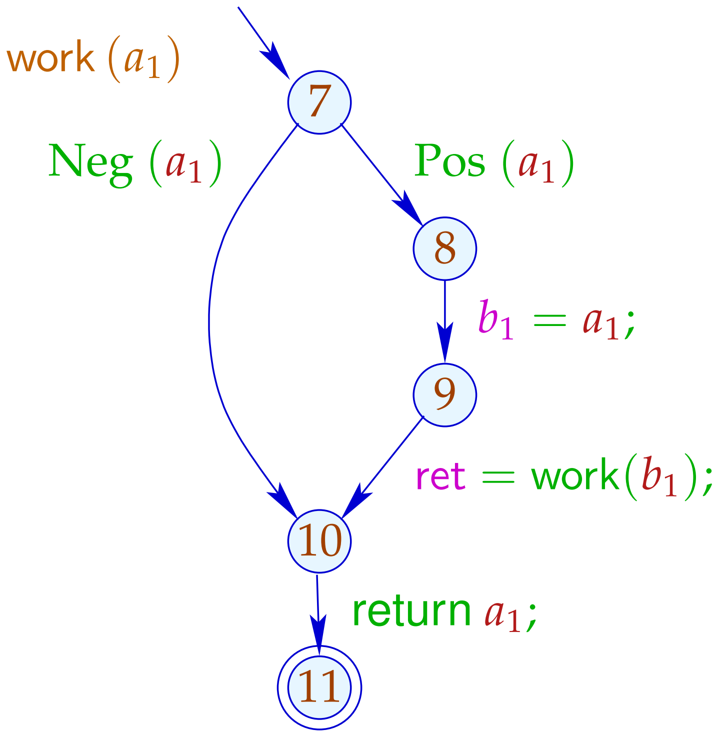
$$
\begin{array}{llcl}
\text{Falls} & [\![\mathsf{work}]\!]^\sharp & = & \{a_1 \mapsto a_1, b_1 \mapsto b_1, \mathsf{ret} \mapsto a_1\} \\
\text{dann} & H_{\mathsf{work}}\,[\![\mathsf{work}]\!]^\sharp & = & \mathsf{Id} \oplus \{\mathsf{ret} \mapsto (([\![\mathsf{work}]\!]^\sharp \circ E_f)\ \mathsf{ret})\} \\
 & & = & \mathsf{Id} \oplus \{\mathsf{ret} \mapsto (\boxed{\{a_1 \mapsto b_1, b_1 \mapsto 0, \mathsf{ret} \mapsto b_1\}}\ \mathsf{ret})\} \\
 & & = & \{a_1 \mapsto a_1, b_1 \mapsto b_1, \mathsf{ret} \mapsto b_1\}
\end{array}
$$

Damit können wir die Fixpunkt-Iteration durchführen :-)



	1
7	$\{a_1 \mapsto a_1, b_1 \mapsto b_1, \mathsf{ret} \mapsto \mathsf{ret}\}$
10	$\{a_1 \mapsto a_1, b_1 \mapsto b_1, \mathsf{ret} \mapsto \mathsf{ret}\}$
11	$\{a_1 \mapsto a_1, b_1 \mapsto b_1, \mathsf{ret} \mapsto a_1\}$
8	$\{a_1 \mapsto a_1, b_1 \mapsto b_1, \mathsf{ret} \mapsto \mathsf{ret}\}$
9	$\{a_1 \mapsto a_1, b_1 \mapsto a_1, \mathsf{ret} \mapsto \mathsf{ret}\}$

$$
\begin{aligned}
[\![(9,\ldots,10)]\!]^\sharp \circ [\![9]\!]^\sharp \quad &= \quad \{a_1 \mapsto a_1, b_1 \mapsto b_1, \mathsf{ret} \mapsto b_1\} \;\circ \\
& \{a_1 \mapsto a_1, b_1 \mapsto a_1, \mathsf{ret} \mapsto \mathsf{ret}\} \\
&= \quad \{a_1 \mapsto a_1, b_1 \mapsto a_1, \mathsf{ret} \mapsto a_1\}
\end{aligned}
$$



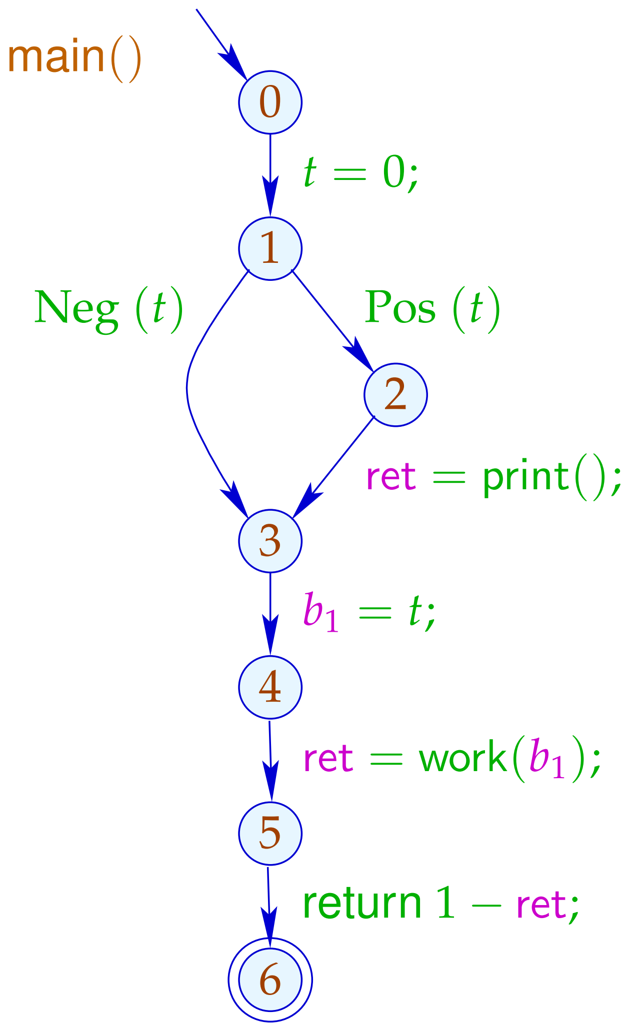
	2
7	$\{a_1 \mapsto a_1, b_1 \mapsto b_1, \mathsf{ret} \mapsto \mathsf{ret}\}$
10	$\boxed{\{a_1 \mapsto a_1, b_1 \mapsto a_1 \sqcup b_1, \mathsf{ret} \mapsto a_1 \sqcup \mathsf{ret}\}}$
11	$\boxed{\{a_1 \mapsto a_1, b_1 \mapsto a_1 \sqcup b_1, \mathsf{ret} \mapsto a_1\}}$
8	$\{a_1 \mapsto a_1, b_1 \mapsto b_1, \mathsf{ret} \mapsto \mathsf{ret}\}$
9	$\{a_1 \mapsto a_1, b_1 \mapsto a_1, \mathsf{ret} \mapsto \mathsf{ret}\}$

$$\begin{aligned}
[\![(9,\ldots,10)]\!]^\sharp \circ [\![9]\!]^\sharp &= \{a_1 \mapsto a_1, b_1 \mapsto b_1, \mathsf{ret} \mapsto b_1\} \circ \\
&\quad\ \{a_1 \mapsto a_1, b_1 \mapsto a_1, \mathsf{ret} \mapsto \mathsf{ret}\} \\
&= \{a_1 \mapsto a_1, b_1 \mapsto a_1, \mathsf{ret} \mapsto a_1\}
\end{aligned}$$

Wenn wir die Effekte von Funktionsaufrufen kennen, können wir ein Constraintsystem aufstellen, um den abstrakten Zustand bei Erreichen eines Punkts ermitteln:

$$
\begin{array}{lll}
\mathcal{R}[\mathsf{main}] & \sqsupseteq & \mathsf{enter}_0\ d_0 \\
\mathcal{R}[f] & \sqsupseteq & \mathsf{enter}^\sharp_f\ (\mathcal{R}[u]) \qquad k = (u, \mathsf{ret} = f\ (a_1, \ldots, a_k);, \_) \quad \text{Aufruf} \\
\mathcal{R}[v] & \sqsupseteq & \mathcal{R}[f] \qquad v \quad \text{Anfangspunkt von} \quad f \\
\mathcal{R}[v] & \sqsupseteq & [\![k]\!]^\sharp\ (\mathcal{R}[u]) \qquad k = (u, \_, v) \quad \text{Kante}
\end{array}
$$

... im Beispiel:



0	$\{b_1 \mapsto 0, \text{ret} \mapsto 0, t \mapsto 0\}$
1	$\{b_1 \mapsto 0, \text{ret} \mapsto 0, t \mapsto 0\}$
2	$\{b_1 \mapsto 0, \text{ret} \mapsto 0, t \mapsto 0\}$
3	$\{b_1 \mapsto 0, \text{ret} \mapsto \top, t \mapsto 0\}$
4	$\{b_1 \mapsto 0, \text{ret} \mapsto \top, t \mapsto 0\}$
5	$\{b_1 \mapsto 0, \text{ret} \mapsto 0, t \mapsto 0\}$
6	$\{b_1 \mapsto 0, \text{ret} \mapsto \top, t \mapsto 0\}$

Diskussion:

- Zumindest Kopier-Konstanten lassen sich interprozedural ermitteln.
- Dazu mussten wir Bedingungen und kompliziertere Zuweisungen ignorieren :-(
- In der zweiten Phase hätten wir allerdings exakter rechnen können :-)
- Die weitere Abstrahierung war aus zwei Gründen notwendig:

  (1) Die Menge der auftretenden Transformer $\mathbb{M} \subseteq \mathbb{D} \to \mathbb{D}$ muss endlich sein;

  (2) Die Funktionen $M \in \mathbb{M}$ müssen effizient implementierbar sein :-)

- Auf die zweite Bedingung kann evt. verzichtet werden ...

Beobachtung: Sharir/Pnueli, Cousot

- → Oft werden Funktionen nur mit wenigen verschiedenen abstrakten Argumenten aufgerufen.
- → Man könnte dann doch jede Funktion für nur genau diese Aufrufe analysieren :-)
- → Stelle das folgende Constraint-System auf:

$$
\begin{array}{lll}
[\![v, a]\!]^\sharp & \sqsupseteq & a \qquad\qquad\qquad v \quad \text{Eintrittspunkt} \\
[\![v, a]\!]^\sharp & \sqsupseteq & \mathsf{combine}^\sharp\,([\![u, a]\!], [\![f, \mathsf{enter}_f^\sharp\,[\![u, a]\!]^\sharp]\!]^\sharp) \\
 & & \qquad\qquad (u, \mathsf{ret} = f\,(a_1, \ldots, a_k);, v) \quad \text{Aufruf} \\
[\![v, a]\!]^\sharp & \sqsupseteq & [\![lab]\!]^\sharp\,[\![u, a]\!]^\sharp \quad k = (u, lab, v) \quad \text{Kante} \\
[\![f, a]\!]^\sharp & \sqsupseteq & [\![stop_f, a]\!]^\sharp \qquad stop_f \quad \text{Endpunkt von} \quad f
\end{array}
$$

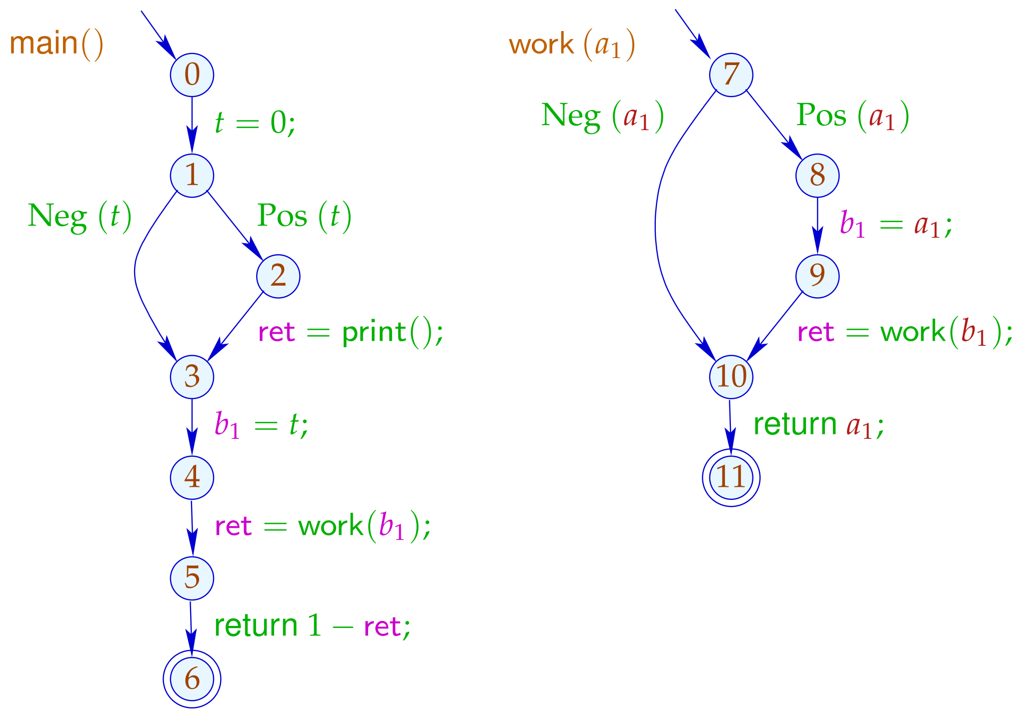
// $[\![v, a]\!]^\sharp$ == Wert des Effekts für das Argument a .

Diskussion:

- Dieses Constraint-System ist i.a. riesengroß :-(
- Wir wollen es aber gar nicht komplett lösen !!!
- Uns reicht es, die korrekten Werte für alle Aufrufe zu ermitteln, die vorkommen, d.h. für die Berechnung des Werts $[\![\mathsf{main}(), a_0]\!]^\sharp$ benötigt werden $\Longrightarrow$ wir verwenden unseren lokalen Fixpunkt-Algorithmus :-))
- Der Fixpunkt-Algo liefert uns sogar noch die Menge der aktuellen Parameter $a \in \mathbb{D}$, für die eine Funktion (möglicherweise) aufgerufen wird sowie die Werte an allen ihren Programm-Punkten für jeden dieser Aufrufe :-)

... im Beispiel:

Versuchen wir einfach einmal eine volle Konstanten-Propagation ...



0	$\lambda x.0$	$\lambda x.0$
1	$\lambda x.0$	$\lambda x.0$
2	$\lambda x.0$	$\bot$
3	$\lambda x.0$	$\lambda x.0$
4	$\lambda x.0$	$\lambda x.0$
7	$\lambda x.0$	$\lambda x.0$
8	$\lambda x.0$	$\bot$
9	$\lambda x.0$	$\bot$
10	$\lambda x.0$	$\lambda x.0$
11	$\lambda x.0$	$\lambda x.0$
5	$\lambda x.0$	$\lambda x.0$
6	$\lambda x.0$	$(\lambda x.0) \oplus \{\text{ret} \mapsto 1\}$
main()	$\lambda x.0$	$(\lambda x.0) \oplus \{\text{ret} \mapsto 1\}$

Diskussion:

- Im Beispiel terminiert die Analyse schnell :-)
- Falls $\mathbb{D}$ endliche Höhe hat, terminiert die Analyse, sofern nur jede Funktion während der Iteration nur mit endlich vielen verschiedenen Argumenten aufgerufen wird :-))
- Analoge Analyse-Algorithmen erwiesen sich bei der Analyse von Prolog als äußerst effizient und präzise :-)
- Zusammen mit einer Points-To-Analyse und Propagation selbst von negativer Konstanten-Information haben wir diesen Algorithmus äußerst erfolgreich zur Fehlersuche in C mit Posix-Threads eingesetzt :-)

(2) Der Call-String-Ansatz:

Idee:

$\rightarrow$ Berechne die Menge aller erreichbaren Aufrufkeller!

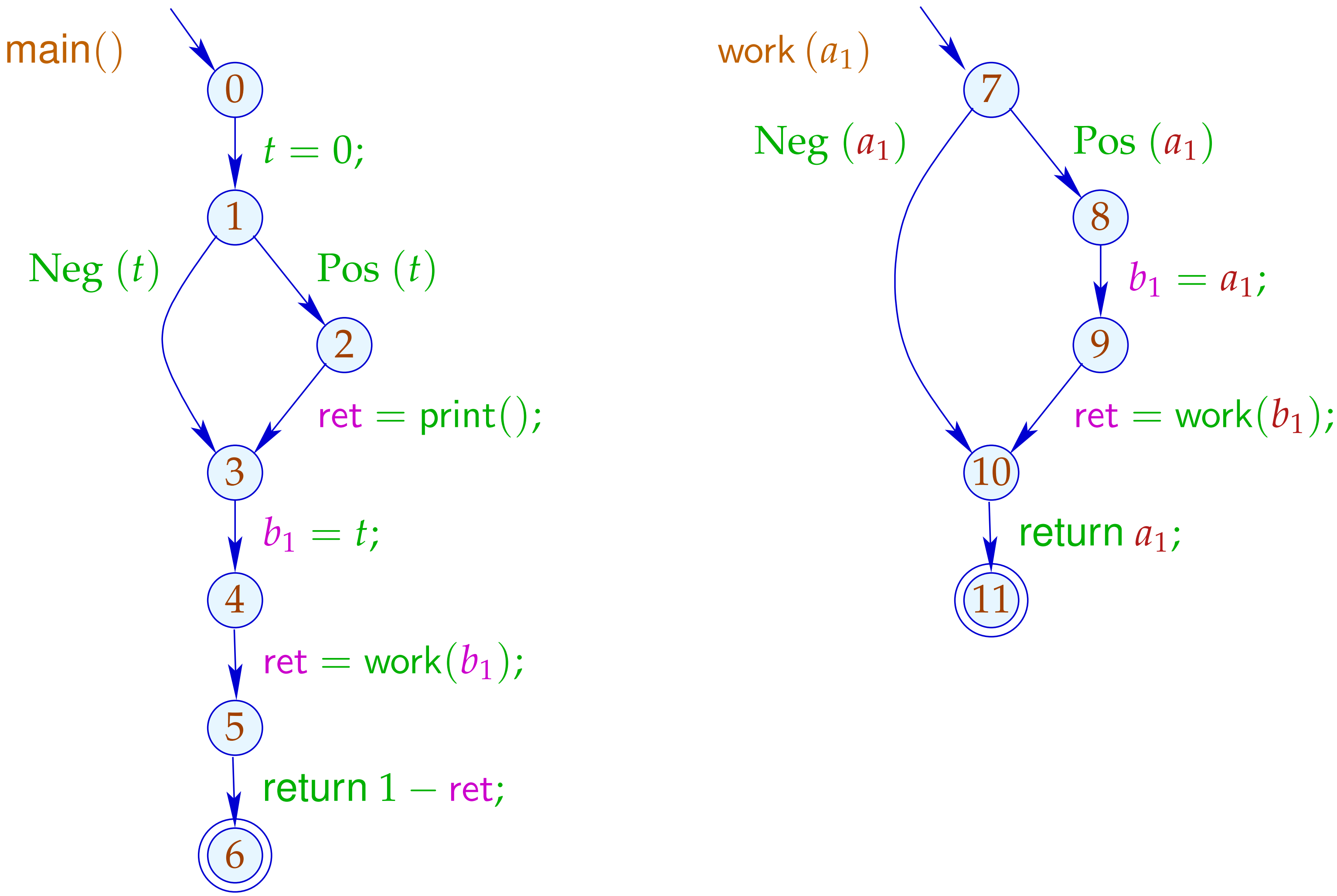
$\rightarrow$ Diese ist i.a. unendlich :-(

$\rightarrow$ Behandle Keller bis zu einer festen Tiefe d exakt! Behalte von längeren Kellern nur das obere Ende der Länge d :-)

$\rightarrow$ Wichtiger Spezialfall: $d = 0$.

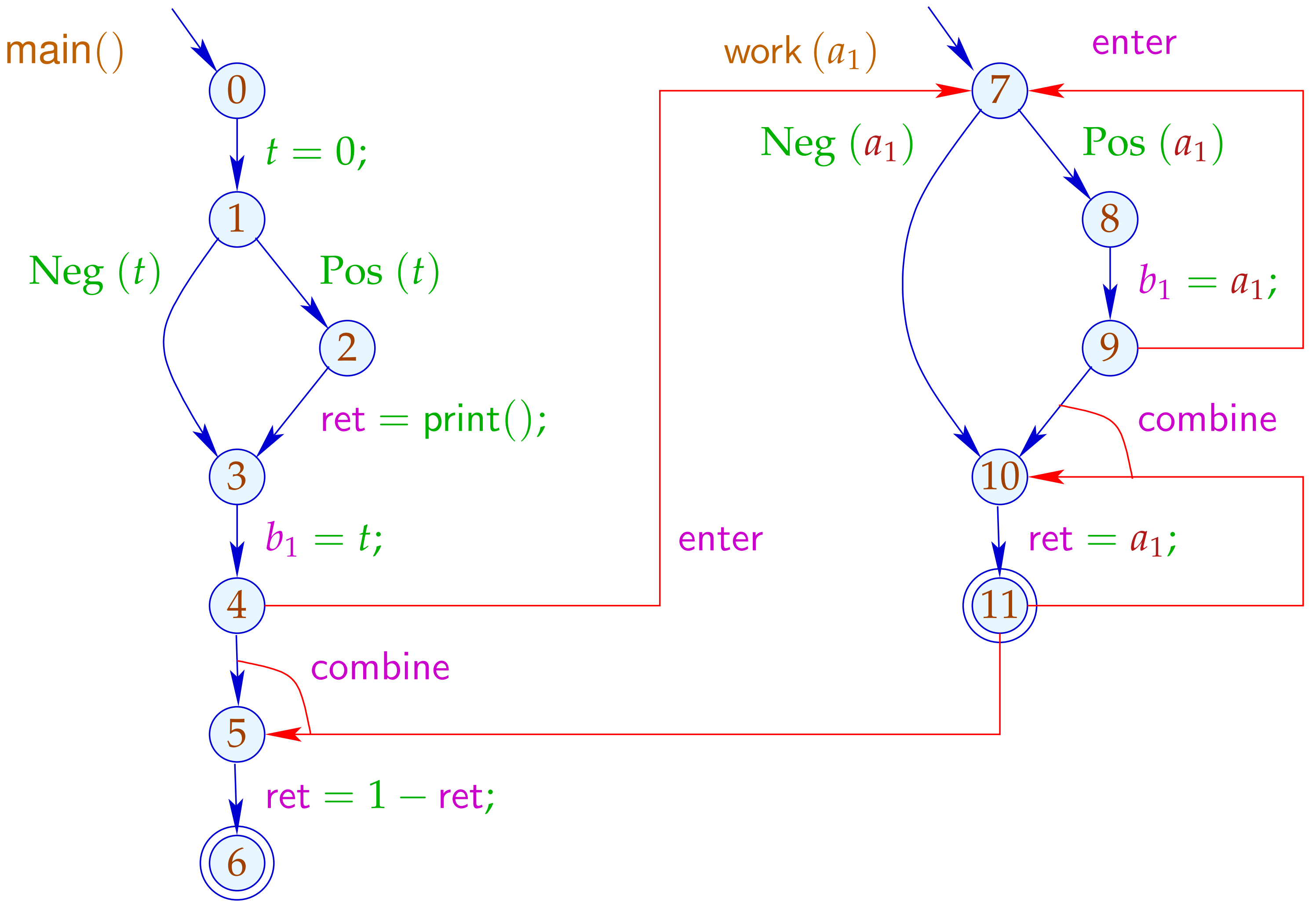
$\Longrightarrow$ Betrachte nur die obersten Kellerrahmen ...

... im Beispiel:



main()
0
$t = 0;$
1
Neg (t)
Pos (t)
2
ret = print();
3
$b_1 = t;$
4
ret = work(b_1);
5
return 1 − ret;
6
work (a_1)
7
Neg (a_1)
Pos (a_1)
8
$b_1 = a_1;$
9
ret = work(b_1);
10
return a_1;
11

... im Beispiel:



main()
0
$t = 0;$
1
Neg (t)
Pos (t)
2
ret = print();
3
$b_1 = t;$
4
combine
5
ret = 1 − ret;
6
work (a_1)
enter
7
Neg (a_1)
Pos (a_1)
8
$b_1 = a_1;$
9
combine
10
enter
ret $= a_1;$
11

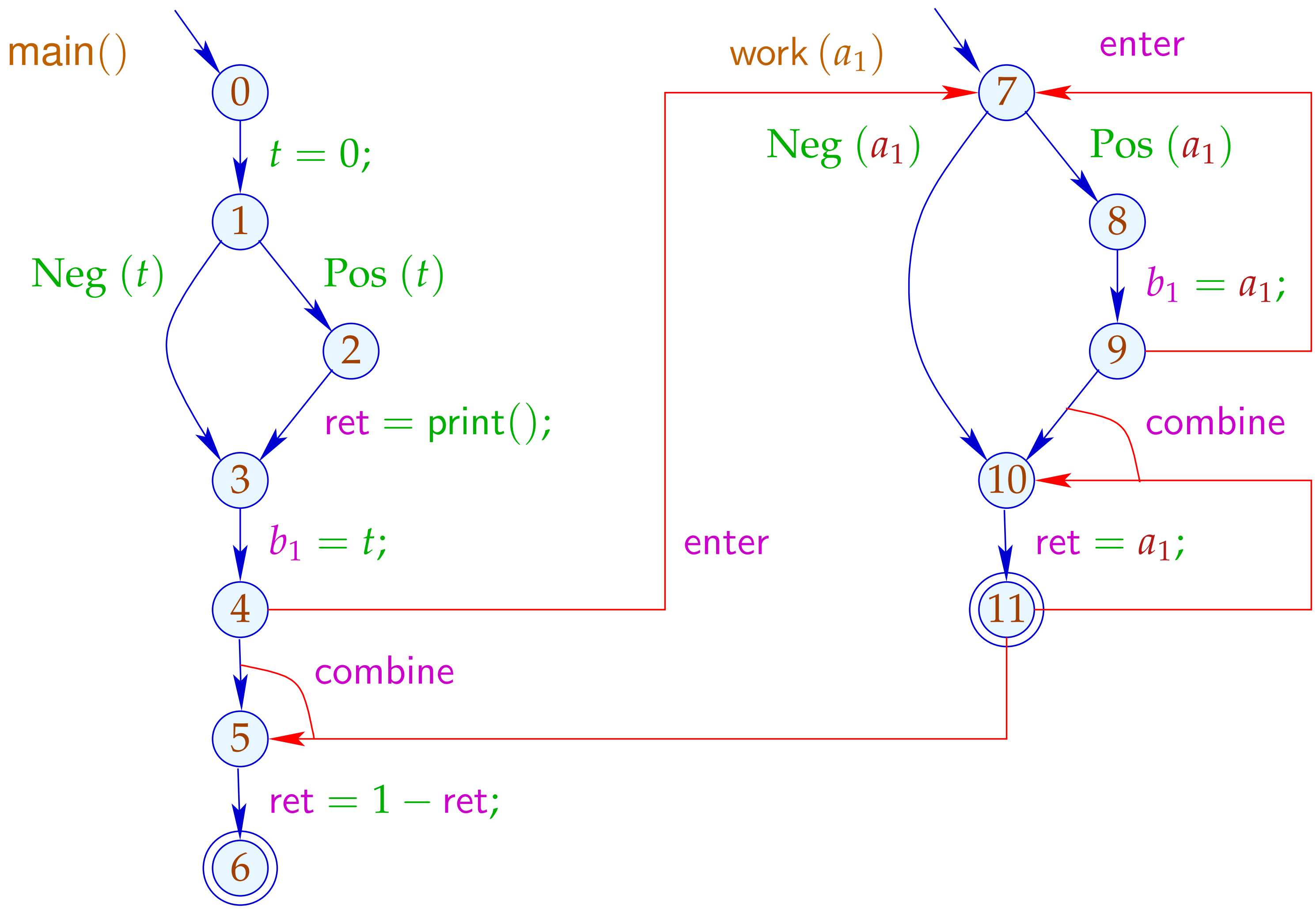
Die Bedingungen für $5, 7, 10$ sind dann etwa:

$$
\begin{array}{lll}
\mathcal{R}[5] & \sqsupseteq & \mathsf{combine}^\sharp\,(\mathcal{R}[4], \mathcal{R}[11]) \\[1ex]
\mathcal{R}[7] & \sqsupseteq & \mathsf{enter}_f^\sharp\,(\mathcal{R}[4]) \\
\mathcal{R}[7] & \sqsupseteq & \mathsf{enter}_f^\sharp\,(\mathcal{R}[9]) \\[1ex]
\mathcal{R}[10] & \sqsupseteq & \mathsf{combine}^\sharp\,(\mathcal{R}[9], \mathcal{R}[11])
\end{array}
$$

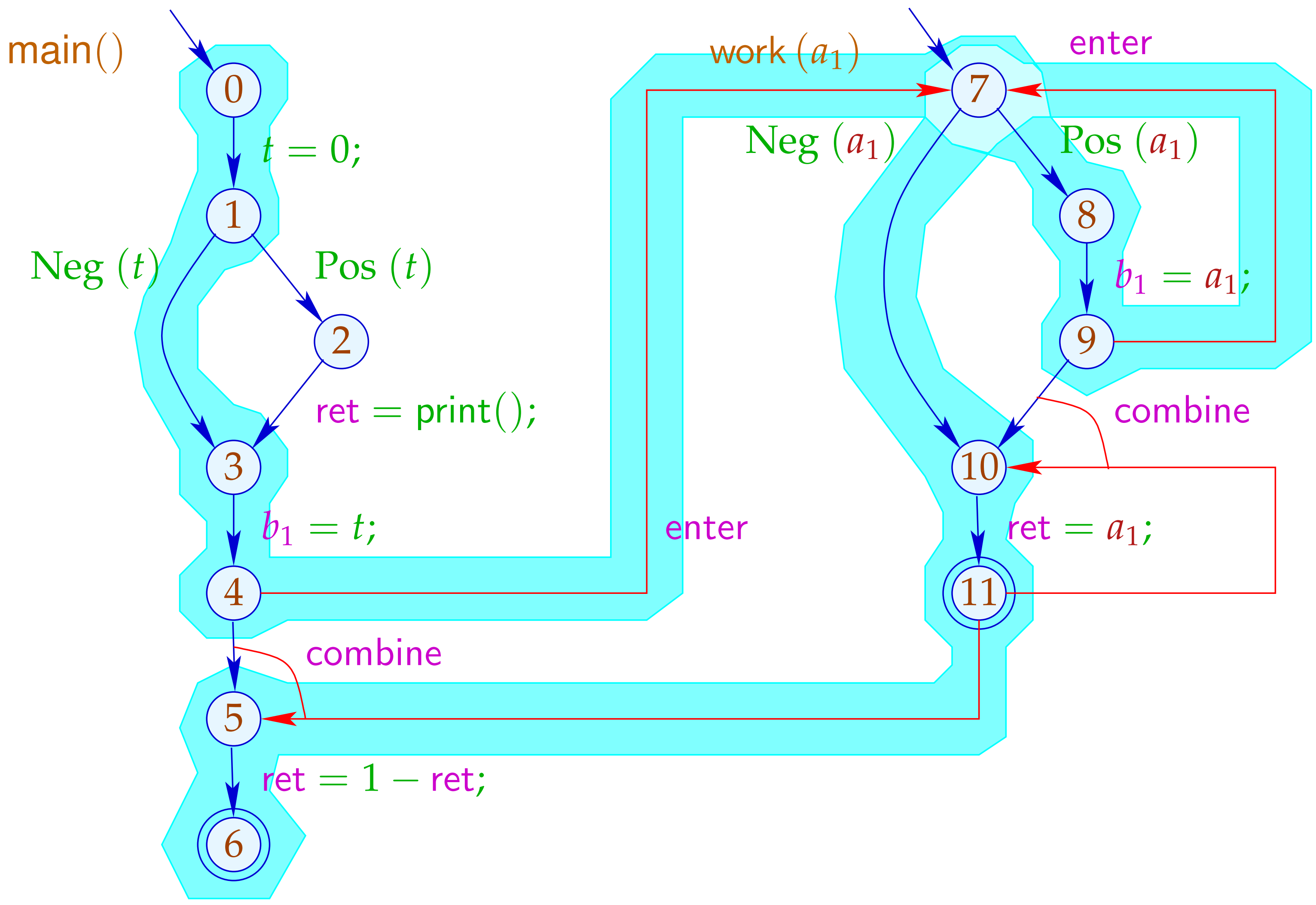
Achtung:

Der resultierende Supergraph enthält offensichtlich unmögliche Pfade ...

... im Beispiel ist das etwa:



... im Beispiel ist das etwa:



Beachte:

→ Im Beispiel finden wir zwar die gleichen Ergebnisse: Mehr Pfade machen die Ergebnissse evt. weniger präzise.

Insbesondere analysieren wir jede Funktion nur für ein (evt. sehr nichtssagendes) Argument-Tupel :-(

→ Die Analyse terminiert — sofern nur $\mathbb{D}$ keine unendlichen echt aufsteigenden Ketten besitzt :-)

→ Die Korrektheit zeigt man relativ zur operationellen Semantik mit den Stacks.

→ Für die Korrektheit des funktionalen Ansatzes ist die Semantik über Berechnungswälder besser geeignet :-)

3 Ausnutzung von Hardware-Einrichtungen

Frage: Wie nutzt man optimal

... Register

... Instruktionen

... Pipelines

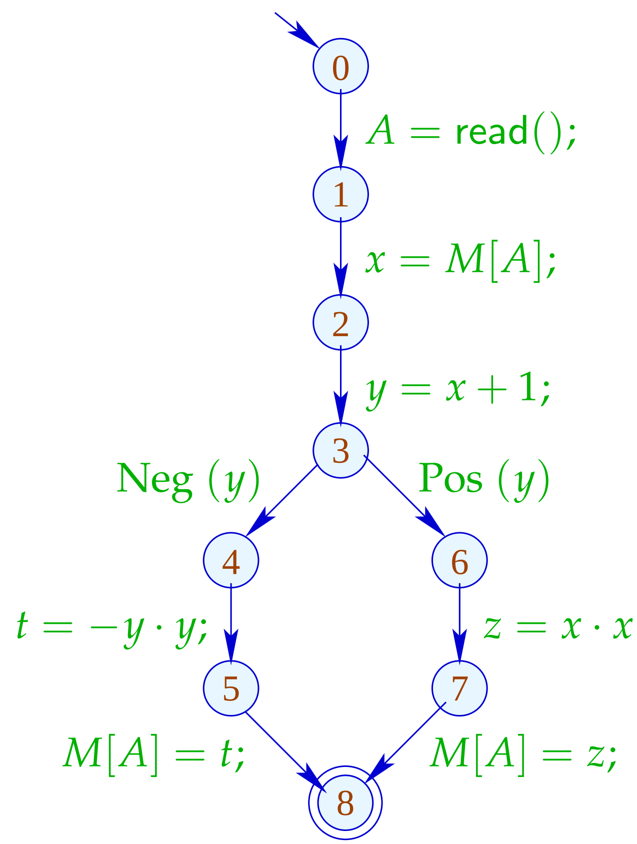
... Caches

... Prozessoren ???

3.1 Register

Beispiel:

$A = \mathsf{read}();$
$x = M[A];$
$y = x + 1;$
$\mathsf{if}\ \ (y)\ \ \{$
$\qquad z = x \cdot x;$
$\qquad M[A] = z;$
$\}\ \ \mathsf{else}\ \ \{$
$\qquad t = -y \cdot y;$
$\qquad M[A] = t;$
$\}$



Das Programm benötigt 5 Variablen ...

Problem:

Was tun, wenn das Programm benutzt mehr Variablen als Register da sind :-(

Idee:

Benutze ein Register für mehrere Variablen :-)

Im Beispiel etwa eines für x, t, z ...

$A = \mathsf{read}();$

$x = M[A];$

$y = x + 1;$

$\mathsf{if}\ \ (y)\ \ \{$

$\qquad z = x \cdot x;$

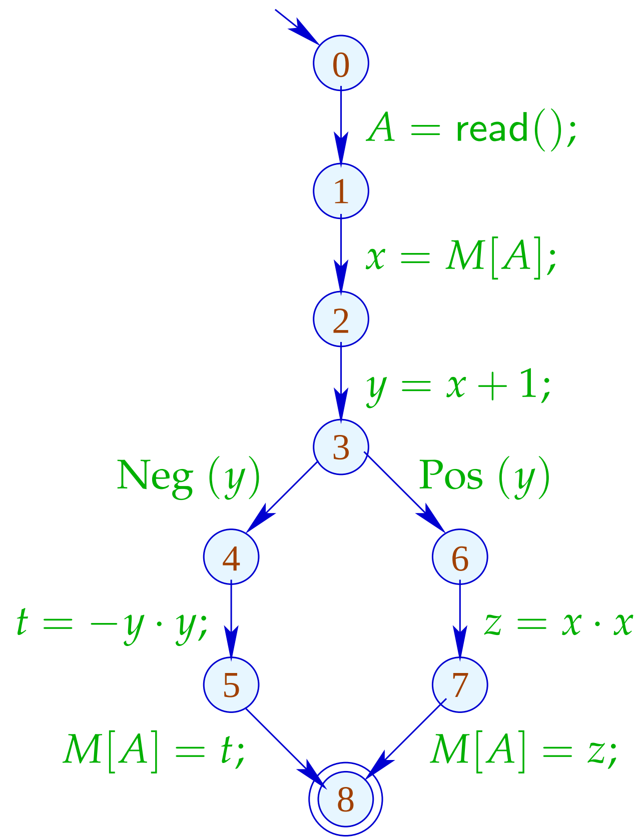
$\qquad M[A] = z;$

$\}\ \ \mathsf{else}\ \ \{$

$\qquad t = -y \cdot y;$

$\qquad M[A] = t;$

$\}$



$A = \mathsf{read}();$

$R = M[A];$

$y = R + 1;$

$\mathsf{if}\ \ (y)\ \ \{$

$\qquad R = R \cdot R;$

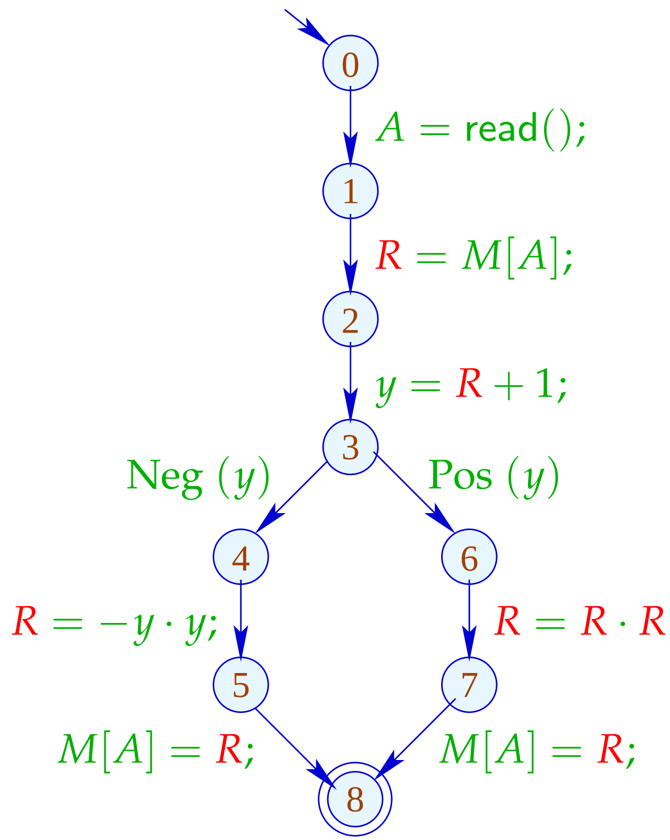
$\qquad M[A] = R;$

$\}\ \ \mathsf{else}\ \ \{$

$\qquad R = -y \cdot y;$

$\qquad M[A] = R;$

$\}$



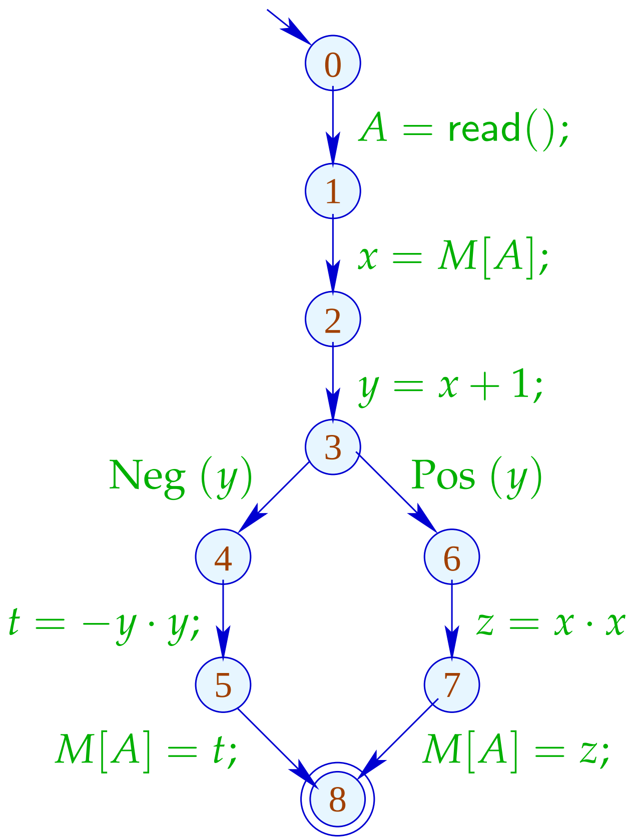
Achtung:

Das geht nur, wenn sich die Lebendigkeitsbereiche nicht überschneiden :-)

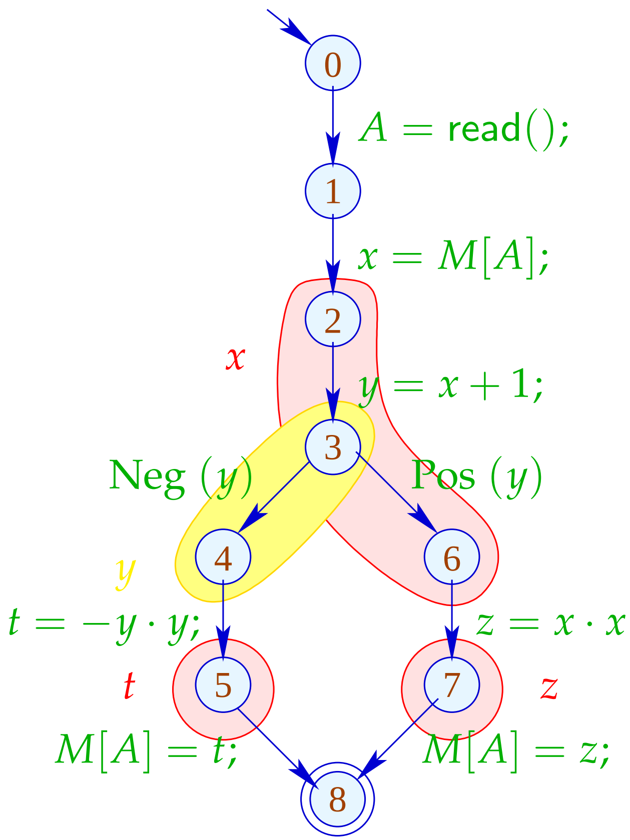
Der (wahre) Lebendigkeitsbereich von x ist:

$$\mathcal{L}[x] \;=\; \{u \mid x \in \mathcal{L}[u]\}$$

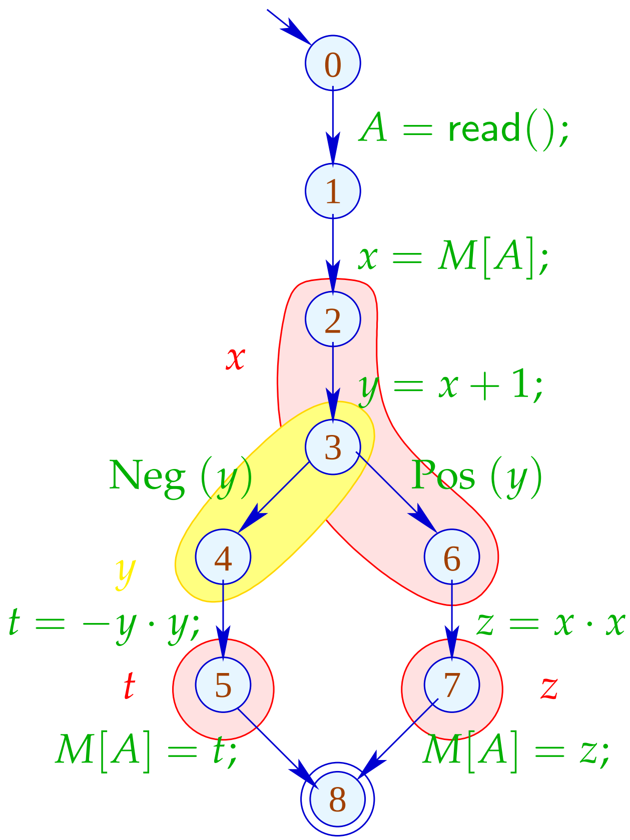
... im Beispiel:



	$\mathcal{L}$
8	$\emptyset$
7	$\{A, z\}$
6	$\{A, x\}$
5	$\{A, t\}$
4	$\{A, y\}$
3	$\{A, x, y\}$
2	$\{A, x\}$
1	$\{A\}$
0	$\emptyset$



	$\mathcal{L}$
8	$\emptyset$
7	$\{A, z\}$
6	$\{A, x\}$
5	$\{A, t\}$
4	$\{A, y\}$
3	$\{A, x, y\}$
2	$\{A, x\}$
1	$\{A\}$
0	$\emptyset$



Lebendigkeitsbereiche:

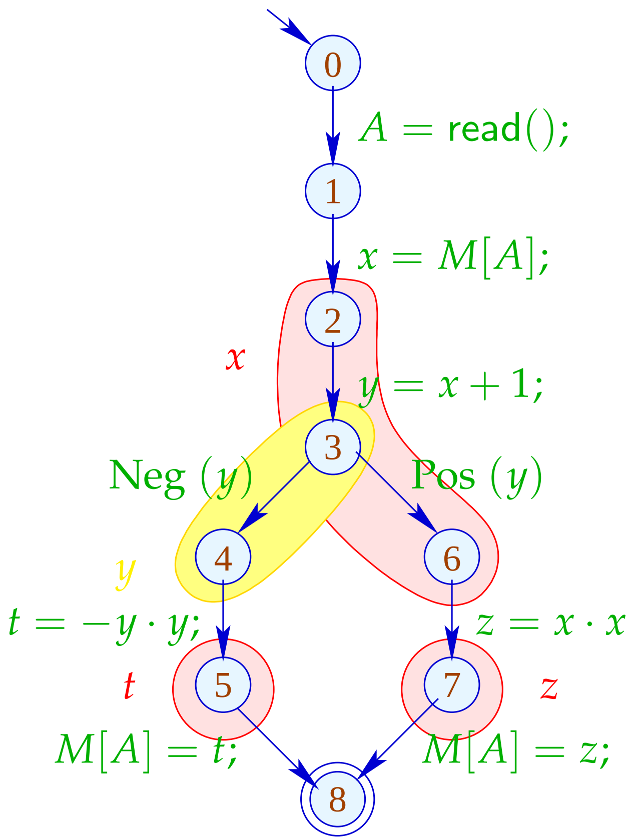
A	$\{1,\ldots,7\}$
x	$\{2,3,6\}$
y	$\{2,4\}$
t	$\{5\}$
z	$\{7\}$

Um Mengen kompatibler Variablen zu finden, konstruieren wir den Interferenz-Graphen $I = (Vars, E_I)$, wobei:

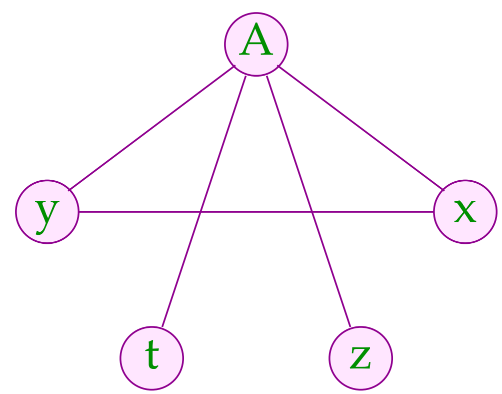
$$E_I = \{\{x, y\} \mid x \neq y, \mathcal{L}[x] \cap \mathcal{L}[y] \neq \emptyset\}$$

E_I enthält eine Kante für $x \neq y$ genau dann wenn x, y an einem gemeinsamen Punkt lebendig sind :-)

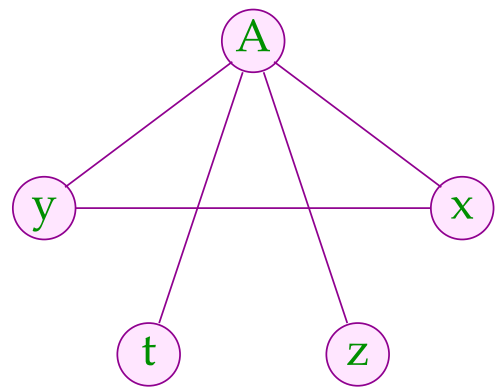
... im Beispiel:



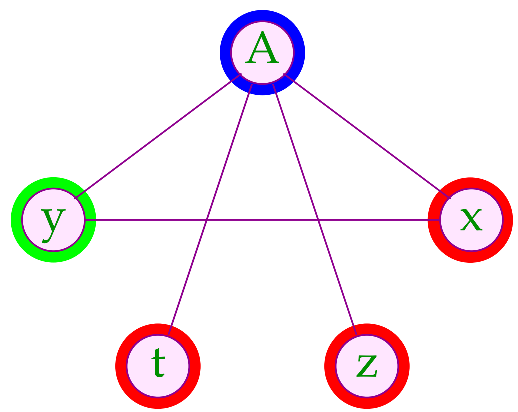
Interferenz-Graph:



Variablen, die nicht mit einer Kante verbunden sind, dürfen dem gleichen Register zugeordnet werden :-)

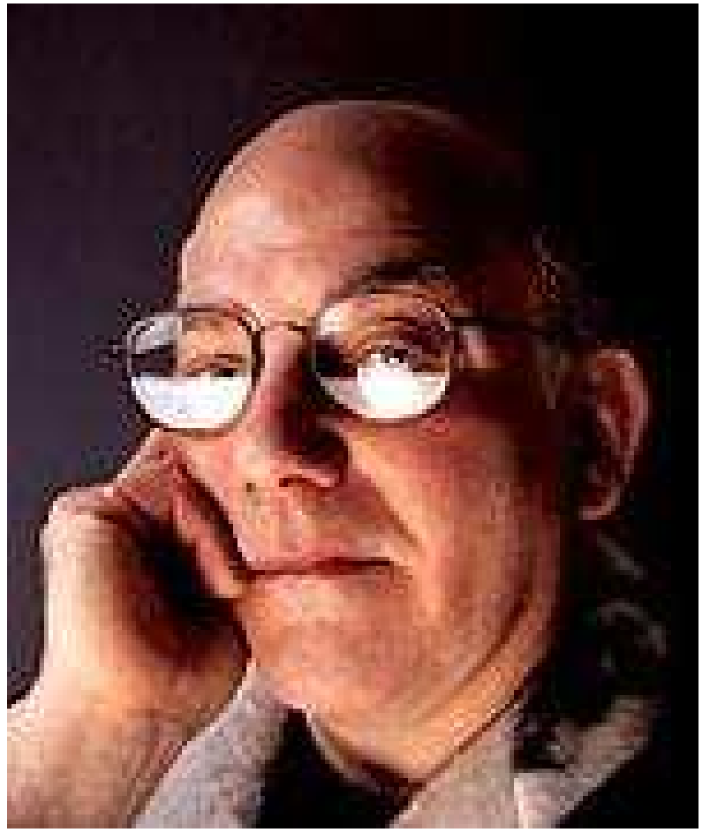


Variablen, die nicht mit einer Kante verbunden sind, dürfen dem gleichen Register zugeordnet werden :-)



Farbe $=$ Register

Sviatoslav Sergeevich Lavrov,
Russische Akademie der Wissenschaften (1962)



Gregory J. Chaitin, University of Maine (1981)

Abstraktes Problem:

Gegeben: Ungerichteter Graph (V, E) .

Gesucht: Minimale Färbung, d.h. Abbildung $c : V \to \mathbb{N}$ mit

(1) $c(u) \neq c(v)$ für $\{u, v\} \in E$;

(2) $\bigsqcup\{c(u) \mid u \in V\}$ minimal!

- Im Beispiel reichen 3 Farben :-) Aber Achtung:
- Die minimale Färbung ist i.a. nicht eindeutig :-(
- Es ist NP-vollständig herauszufinden, ob eine Färbung mit maximal k Farben möglich ist :-((

$\Longrightarrow$

Wir sind auf Heuristiken angewiesen oder Spezialfälle :-)

Greedy-Heuristik:

- Beginne irgendwo mit der Farbe 1;
- Wähle als jeweils neue Farbe die kleinste Farbe, die verschieden ist von allen bereits gefärbten Nachbarn;
- Ist ein Knoten gefärbt, färbe alle noch nicht gefärbten Nachbarn;
- Behandle eine Zusammenhangskomponente nach der andern ...

... etwas konkreter:

```
forall (v ∈ V) c[v] = 0;
forall (v ∈ V) color (v);

void color (v) {
    if (c[v] ≠ 0) return;
    neighbors = {u ∈ V | {u,v} ∈ E};
    c[v] = ⨅{k > 0 | ∀ u ∈ neighbors : k ≠ c(u)};
    forall (u ∈ neighbors)
        if (c(u) == 0) color (u);
}
```

Die neue Farbe lässt sich leicht berechnen, nachdem die Nachbarn nach ihrer Farbe geordnet wurden :-)

Diskussion:

→ Im wesentlichen ist das Prä-order DFS :-)

→ In der Theorie kann das Ergebnis beliebig weit vom Optimum entfernt sein :-(

→ ... ist aber in der Praxis ganz gut :-)

→ ... Achtung: verschiedene Varianten sind patentiert !!!

Diskussion:

→ Im wesentlichen ist das Prä-order DFS :-)

→ In der Theorie kann das Ergebnis beliebig weit vom Optimum entfernt sein :-(

→ ... ist aber in der Praxis ganz gut :-)

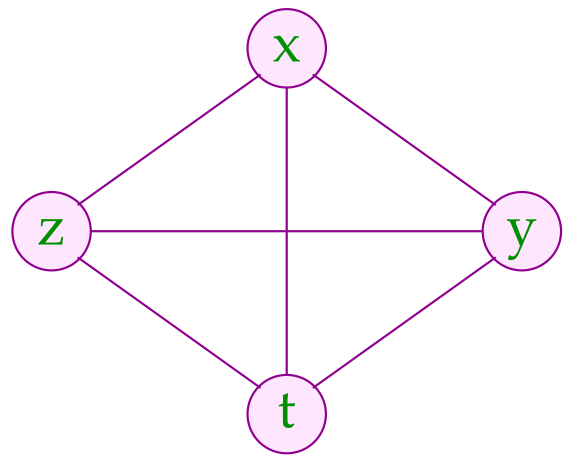
→ ... Achtung: verschiedene Varianten sind patentiert !!!

Der Algorithmus funktioniert umso besser, je kleiner die Lebendigkeitsbereiche sind ...

Idee: Life range splitting

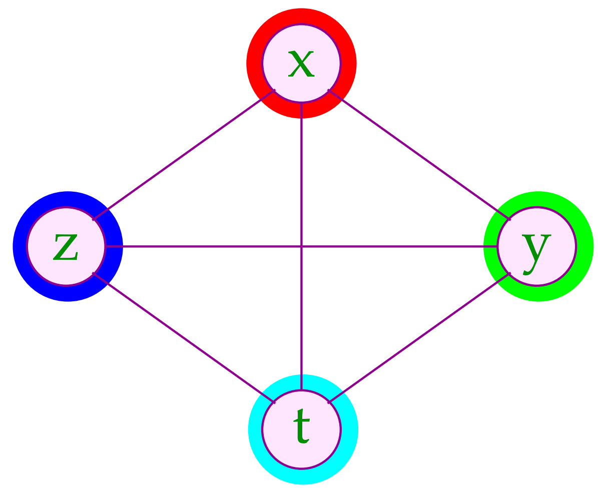
Beispiel:

	$\mathcal{L}$
	x, y, z
$A_1 = x + y;$	x, z
$M[A_1] = z;$	x
$x = x + 1;$	x
$z = M[A_1];$	x, z
$t = M[x];$	x, z, t
$A_2 = x + t;$	x, z, t
$M[A_2] = z;$	x, t
$y = M[x];$	y, t
$M[y] = t;$	



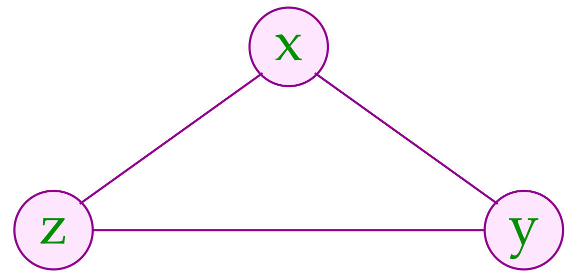
Beispiel:

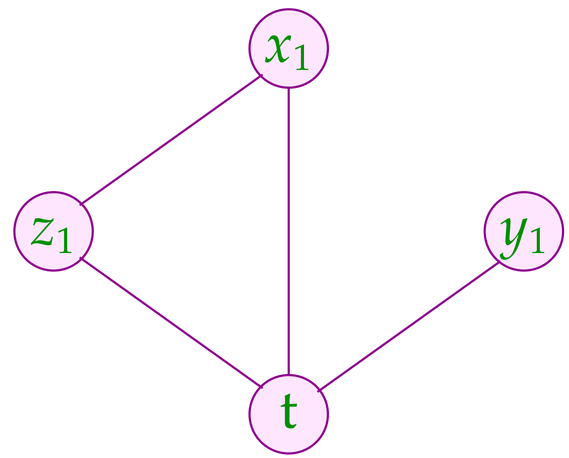
	$\mathcal{L}$
	x, y, z
$A_1 = x + y;$	x, z
$M[A_1] = z;$	x
$x = x + 1;$	x
$z = M[A_1];$	x, z
$t = M[x];$	x, z, t
$A_2 = x + t;$	x, z, t
$M[A_2] = z;$	x, t
$y = M[x];$	y, t
$M[y] = t;$	



Die Lebendigkeitsbereiche von x und z können wir aufteilen:

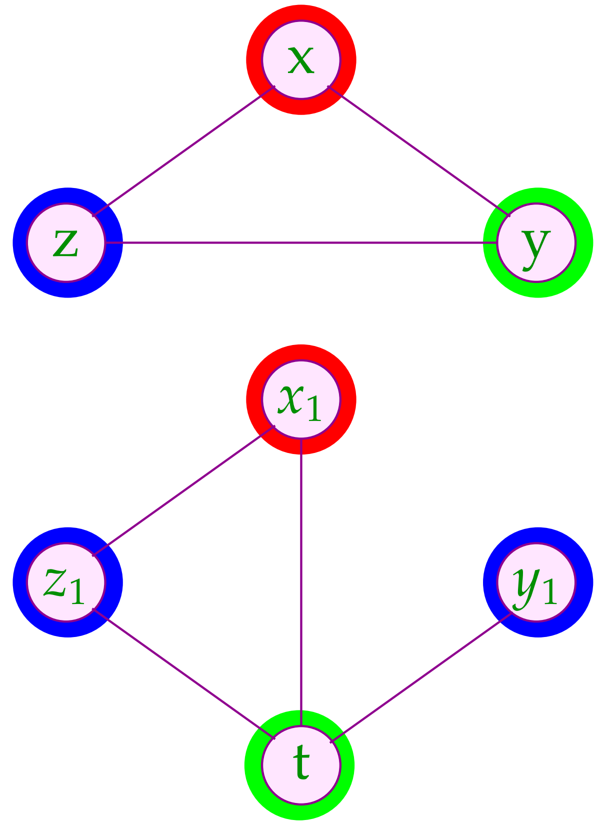
	$\mathcal{L}$
	x, y, z
$A_1 = x + y;$	x, z
$M[A_1] = z;$	x
$x_1 = x + 1;$	x_1
$z_1 = M[A_1];$	x_1, z_1
$t = M[x_1];$	x_1, z_1, t
$A_2 = x_1 + t;$	x_1, z_1, t
$M[A_2] = z_1;$	x_1, t
$y_1 = M[x_1];$	y_1, t
$M[y_1] = t;$	





Die Lebendigkeitsbereiche von x und z können wir aufteilen:

	$\mathcal{L}$
	x, y, z
$A_1 = x + y;$	x, z
$M[A_1] = z;$	x
$x_1 = x + 1;$	x_1
$z_1 = M[A_1];$	x_1, z_1
$t = M[x_1];$	x_1, z_1, t
$A_2 = x_1 + t;$	x_1, z_1, t
$M[A_2] = z_1;$	x_1, t
$y_1 = M[x_1];$	y_1, t
$M[y_1] = t;$	



Technisch:

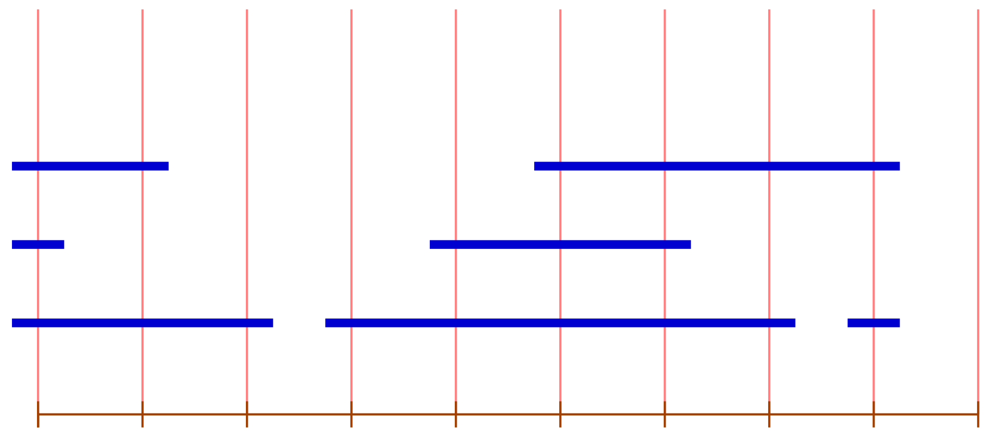
Eine Kante (u, lab, v) heißt x-transparent, falls lab keine Definition von x ist.

u, v gehören zum selben minimalen x-Lebendigkeitsbereich , falls $x \in \mathcal{L}[u] \cap \mathcal{L}[v]$ und u, v durch einen ungerichteten Pfad x-transparenter Kanten verbunden sind ...

Für jeden der minimalen x-Lebendigkeitsbereiche $L_1, \ldots, L_k$ führen wir eine Variante von x ein :-)

Spezialfall: Basis-Blöcke

Die Interferenzgraphen für minimale Lebendigkeitsbereiche auf Folgen von Zuweisungen sind Intervall-Graphen:



Knoten $=$ Intervall

Kante $=$ gemeinsamer Punkt

Zu jedem Punkt können wir die Überdeckungszahl der inzidenten Intervalle angeben.

Satz:

maximale Überdeckungszahl

$=$ Größe der maximalen Klique

$=$ maximal nötige Anzahl Farben :-)

Graphen mit dieser Eigenschaft heißen perfekt ...

Eine minimale Färbung kann in polynomieller Zeit berechnet werden :-))

Idee:

$\longrightarrow$ Iteriere (konzeptuell) über die Punkte $0, \ldots, m-1$!

$\longrightarrow$ Verwalte eine Liste der aktuell freien Farben.

$\longrightarrow$ Beginnt ein neues Intervall, vergib die nächste freie Farbe.

$\longrightarrow$ Endet ein Intervall, gib seine Farbe frei.

Damit ergibt sich folgender Algorithmus:

$$
\begin{array}{l}
\mathsf{free} = [1, \ldots, k]; \\
\mathsf{for}\ \ (i = 0; i < m; i{+}{+})\ \ \{ \\
\qquad \mathsf{init}[i] = []; \ \ \mathsf{exit}[i] = []; \\
\} \\
\mathsf{forall}\ \ (I = [u, v] \in \mathsf{Intervals})\ \ \{ \\
\qquad \mathsf{init}[u] = (I :: \mathsf{init}[u]); \ \ \mathsf{exit}[i] = (I :: \mathsf{exit}[v]); \\
\} \\
\qquad \mathsf{for}\ \ (i = 0; i < m; i{+}{+})\ \ \{ \\
\qquad \mathsf{forall}\ \ (I \in \mathsf{exit}[i])\ \ \mathsf{free} = \mathsf{color}[I] :: \mathsf{free}; \\
\qquad \mathsf{forall}\ \ (I \in \mathsf{init}[i])\ \ \{ \\
\qquad\qquad \mathsf{color}[I] = \mathsf{hd}\ \mathsf{free}; \ \ \mathsf{free} = \mathsf{tl}\ \mathsf{free}; \\
\qquad \} \\
\}
\end{array}
$$

Diskussion:

→ Für Basis-Blöcke können wir eine optimale Aufteilung der Variablen auf eine Register ermitteln :-)

→ Das gleiche Problem ist bereits für einfache Schleifen (circular arc graphs) NP-schwierig :-(

→ Für beliebige Programme wird man deshalb eine Heuristik zum Graph-Färben einsetzen ...

→ Dieses Verfahren funktioniert besser, wenn wir die Lebendigkeitsbereiche maximal unterteilen :-)

→ Reicht die Anzahl der realen Register nicht aus, lagert man die überzähligen in einen festen Speicherbereich aus.

→ Man bemüht sich dabei, zumindest die in innersten Schleifen benutzten Variablen in Registern zu halten.

Interprozedurale Registerverteilung:

$\longrightarrow$ Für jede lokale Variable ist ein Eintrag im Kellerrahmen reserviert.

$\longrightarrow$ Vor dem Aufruf einer Funktion müssen die Register in den Kellerrahmen gerettet und danach restauriert werden.

$\longrightarrow$ Gelegentlich gibt es dafür Hardware-Unterstützung :-)

Dann ist ein Aufruf für alle Register transparent.

$\longrightarrow$ Verwalten wir Retten / Restaurieren selbst, können wir ...

- nur Register retten, deren Inhalte nach dem Aufruf noch benötigt werden :-)
- Register erst bei Bedarf restaurieren — und dann evt. in andere Register $\Longrightarrow$ Verkleinerung der Lebendigkeitsbereiche :-)

3.2 Instruktionen

Problem:

- unregelmäßige Instruktionssätze ...
- mehrere Adressierungsarten, die evt. mit arithmetischen Operationen kombiniert werden können;
- Register für unterschiedliche Verwendungen ...

Beispiel: Motorola MC68000

Dieser einfachste Prozessor der 680x0-Reihe besitzt

- 8 Daten- und 8 Adressregister;
- eine Vielzahl von Adressierungsarten ...

Notation	Beschreibung	Semantik
D_n	Datenregister direkt	D_n
A_n	Adressregister direkt	A_n
(A_n)	Adressregister indirekt	$M[A_n]$
$d(A_n)$	Adressregister indirekt mit Displacement	$M[A_n + d]$
$d(A_n, D_m)$	Adressregister indirekt mit Index und Displacement	$M[A_n + D_m + d]$
x	Absolut kurz	$M[x]$
x	Absolut lang	$M[x]$
$\#x$	Unmittelbar	x

- Der MC68000 ist eine 2-Adress-Maschine, d.h. ein Befehl darf maximal 2 Adressierungen enthalten. Die Instruktion:

  $$\mathsf{add}\;\; D_1 \;\; D_2$$

  addiert die Inhalte von D_1 und D_2 und speichert das Ergebnis nach und D_2 :-)
- Die meisten Befehle lassen sich auf Bytes, Wörter (2 Bytes) oder Doppelwörter (4 Bytes) anwenden.

  Das unterscheiden wir durch Anhängen von .B, .W, .D (Default: .W)
- Die Ausführungszeit eines Befehls ergibt sich (i.a.) aus den Kosten der Operation plus den Kosten für die Adressierung der Operanden ...

	Adressierungsart	Byte / Wort	Doppelwort
D_n	Datenregister direkt	0	0
A_n	Adressregister direkt	0	0
(A_n)	Adressregister indirekt	4	8
$d(A_n)$	Adressregister indirekt mit Displacement	8	12
$d(A_n, D_m)$	Adressregister indirekt mit Index und Displacement	10	14
x	Absolut kurz	8	12
x	Absolut lang	12	16
$\#x$	Unmittelbar	4	8

Beispiel:

Die Instruktion: move.B $8(A_1, D_1.\mathsf{W}), D_5$

benötigt: $4 + 10 + 0 = 14$ Zyklen

Alternativ könnten wir erzeugen:

adda	$\#8, A_1$	Kosten:	$8+8+0=16$
adda	$D_1.\mathsf{W}, A_1$	Kosten:	$8+0+0=8$
move.B	$(A_1), D_5$	Kosten:	$4+4+0=8$

mit Gesamtkosten 32 oder:

adda	$D_1.\mathsf{W}, A_1$	Kosten:	$8+0+0=8$
move.B	$8(A_1), D_5$	Kosten:	$4+8+0=12$

mit Gesamtkosten 20 :-)

Achtung:

- Die verschieden Code-Sequenzen sind im Hinblick auf den Speicher und das Ergebnis äquivalent !
- Sie unterscheiden sich im Hinblick auf den Wert des Registers A_1 sowie die gesetzten Bedingungs-Codes !!
- Ein schlauer Instruktions-Selektor muss solche Randbedingungen berücksichtigen :-)

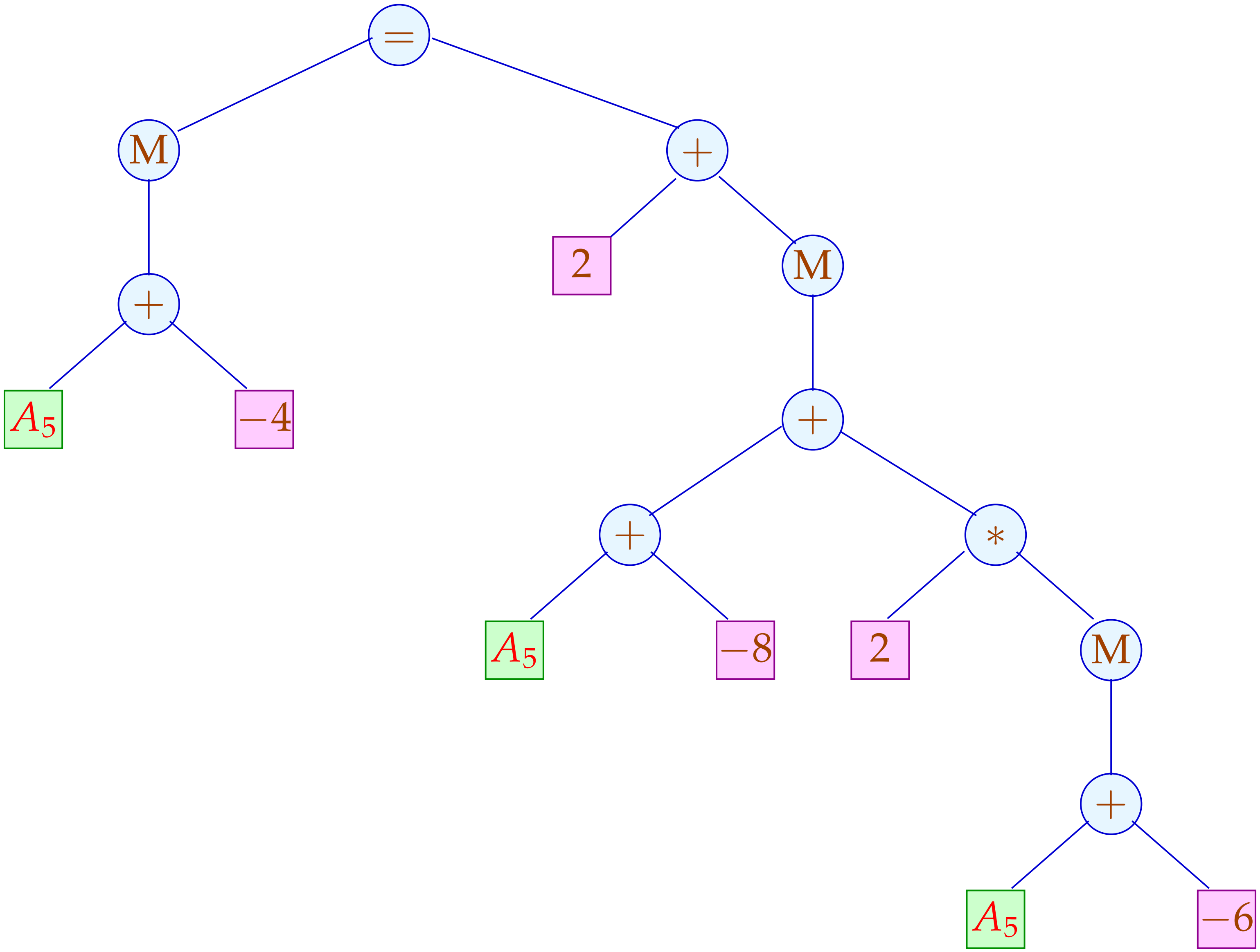
Etwas größeres Beispiel:

$$\text{int } b,\ i,\ a[100];$$
$$b = 2 + a[i];$$

Nehmen wir an, die Variablen werden relativ zu einem Framepointer A_5 mit den Adressen $-4, -6, -8$ adressiert. Dann entspricht der Zuweisung das Stück Zwischen-Code:

$$M[A_5 - 4] = 2 + M[A_5 - 8 + 2 \cdot M[A_5 - 6]];$$

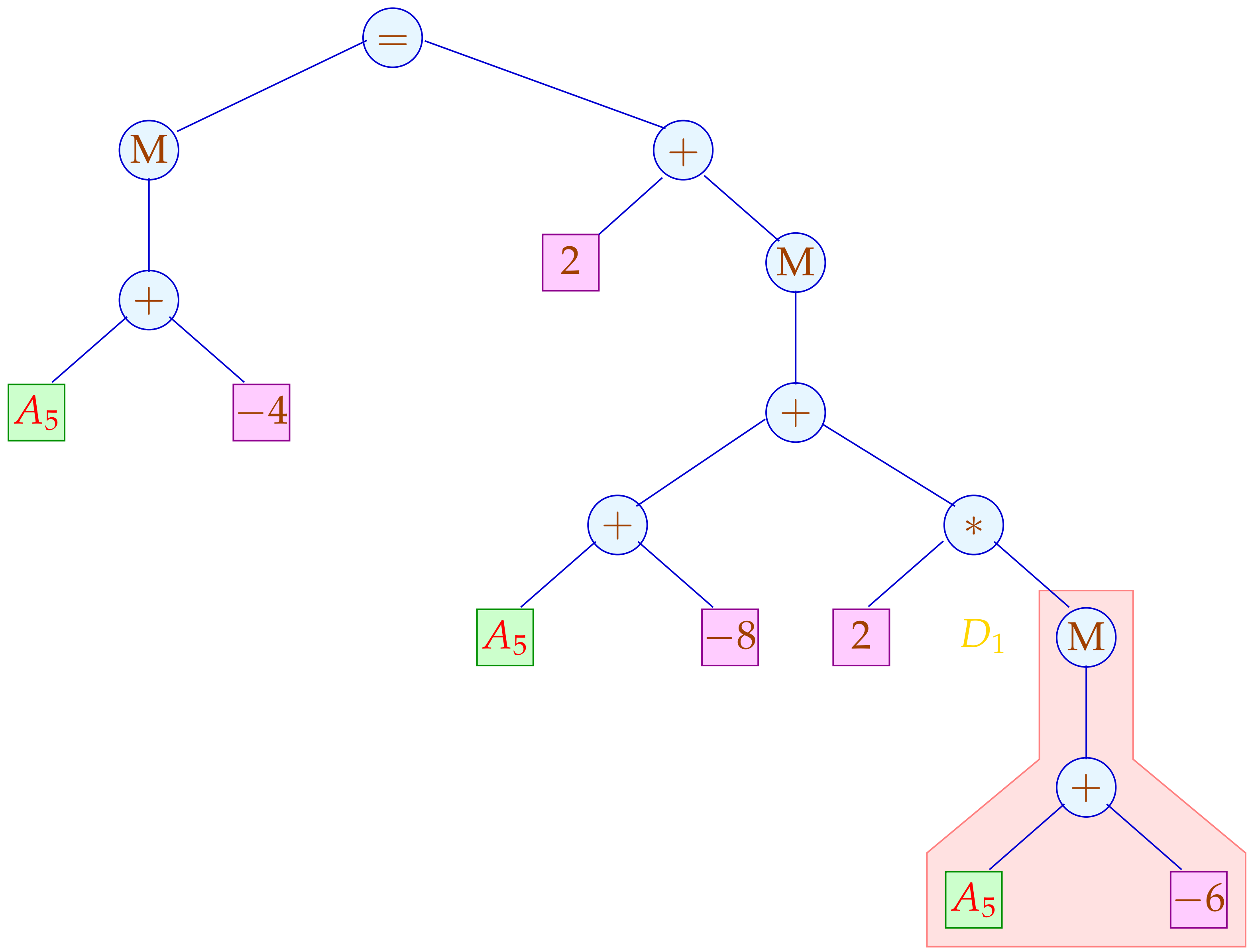
Das entspricht dem Syntaxbaum:

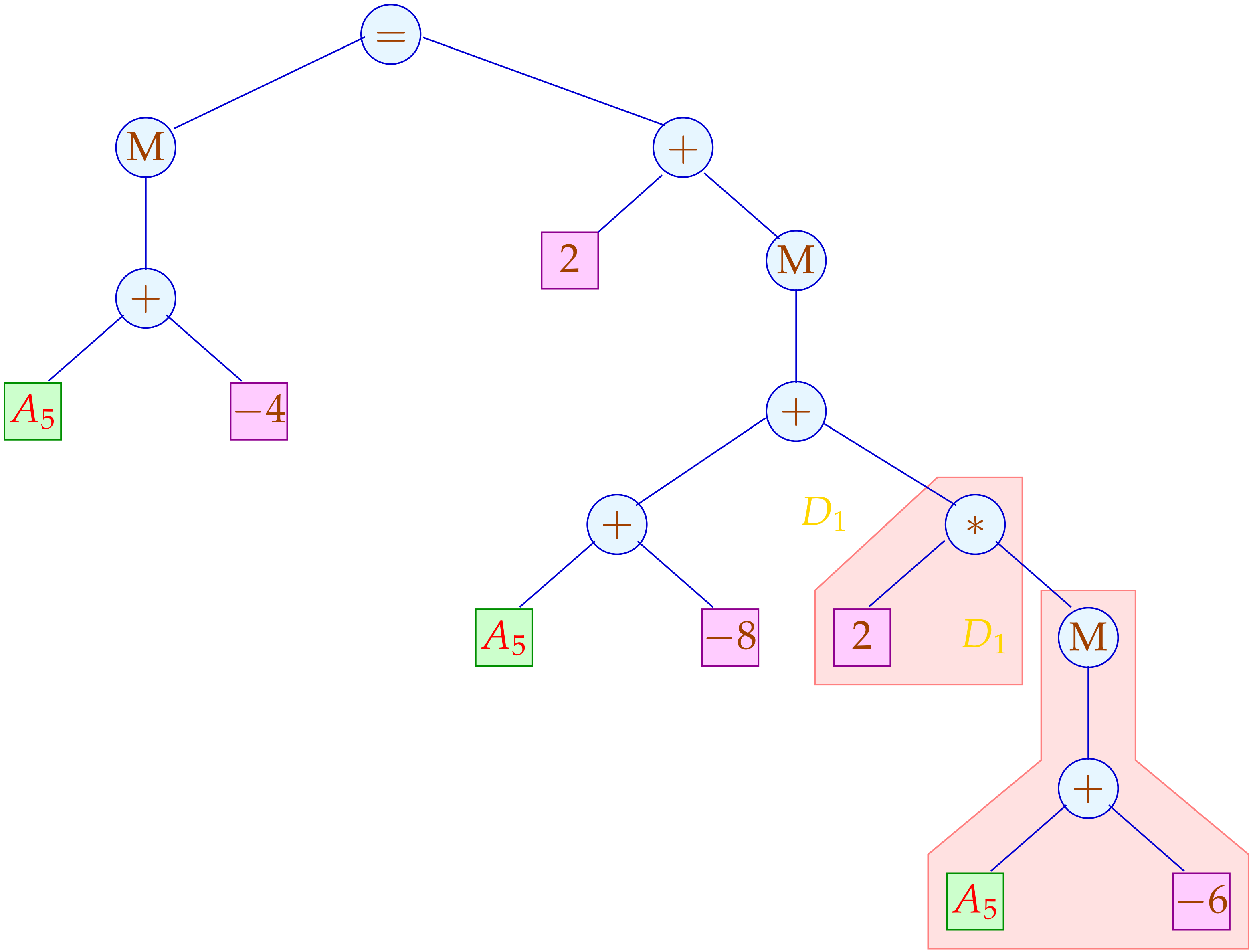

=
M
+
2
M
+
A_5
−4
+
*
A_5
−8
2
M
+
A_5
−6

Eine mögliche Code-Sequenz:

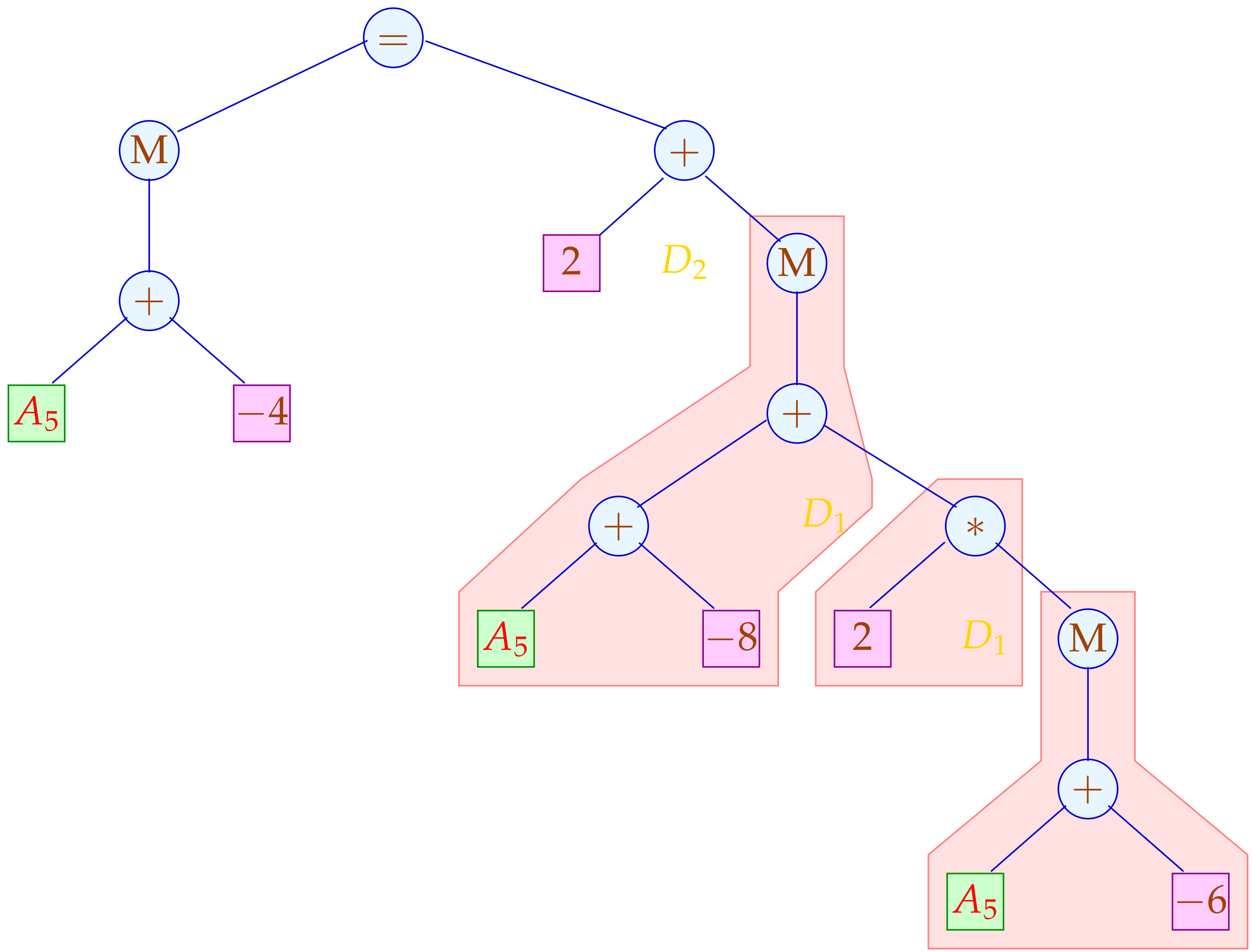
move	$-6(A_5),\ D_1$	Kosten:	12
add	$D_1,\ D_1$	Kosten:	4
move	$-8(A_5, D_1),\ D_2$	Kosten:	14
addq	$\#2,\ D_2$	Kosten:	4
move	$D_2,\ -4(A_5)$	Kosten:	12

Gesamtkosten : 46

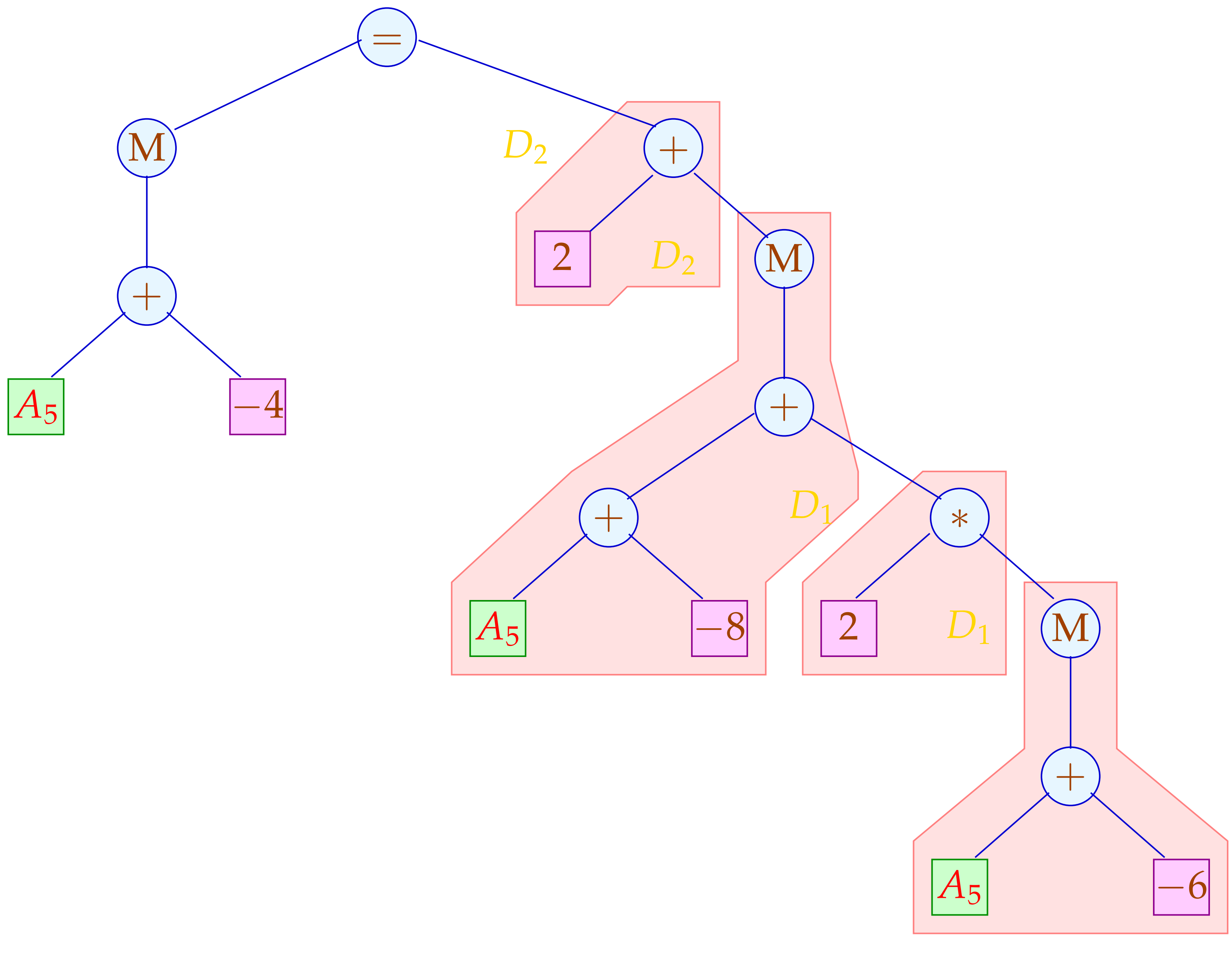

=
M
+
2
M
+
A_5
−4
+
+
∗
A_5
−8
2
D_1
M
+
A_5
−6

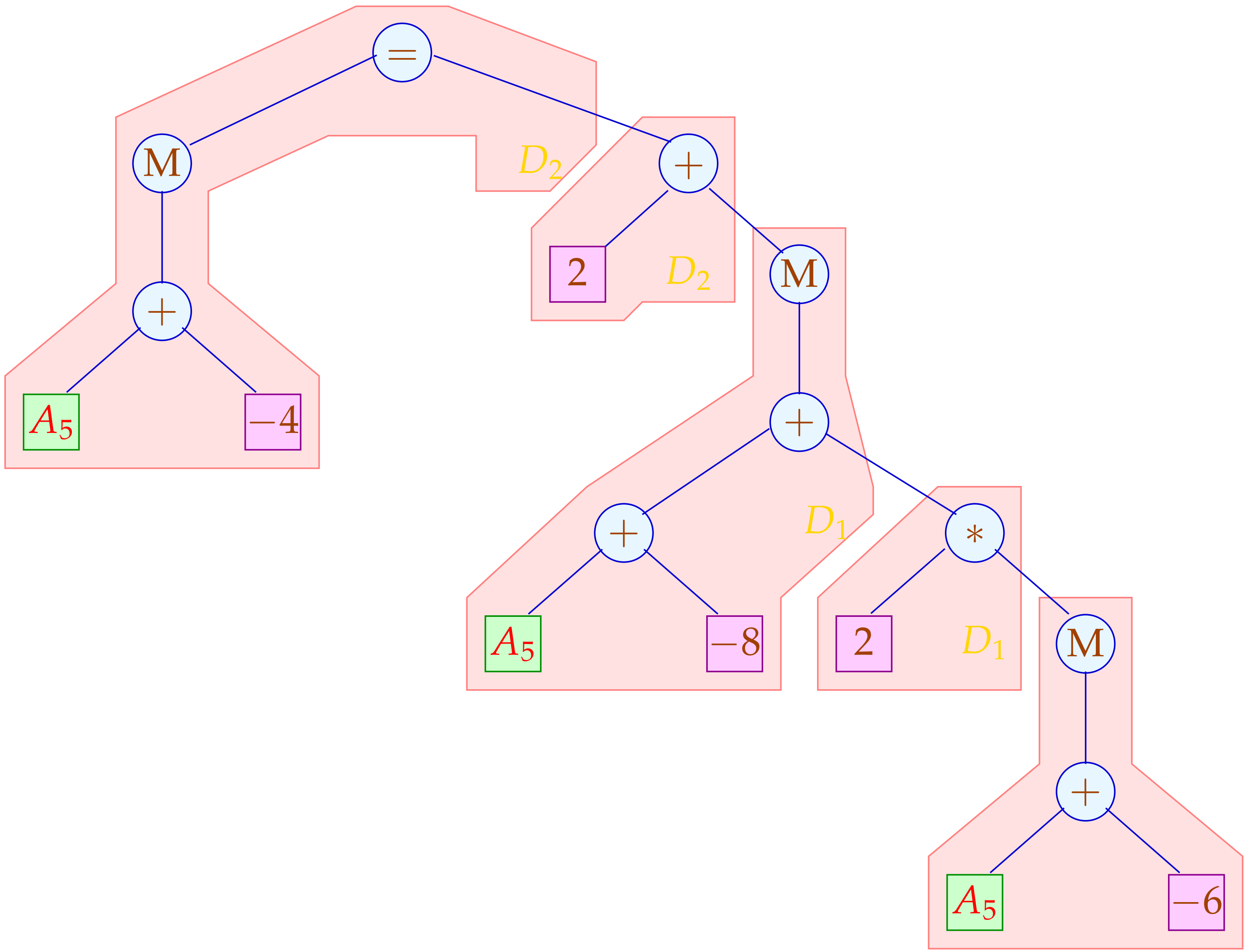


=
M
+
2
M
+
A_5
−4
+
*
D_1
A_5
−8
2
D_1
M
+
A_5
−6



=
M
+
2
D_2
M
+
A_5
−4
+
*
D_1
A_5
−8
2
D_1
M
+
A_5
−6


=
M
+
A_5
−4
D_2
+
2
D_2
M
+
+
A_5
−8
D_1
*
2
D_1
M
+
A_5
−6


=
M
+
A_5
-4
D_2
+
2
D_2
M
+
+
A_5
-8
D_1
*
2
D_1
M
+
A_5
-6

Eine alternative Code-Sequenz:

move.L	$A_5,\ A_1$	Kosten:	4
adda.L	$\#-6,\ A_1$	Kosten:	12
move	$(A_1),\ D_1$	Kosten:	8
mulu	$\#2,\ D_1$	Kosten:	44
move.L	$A_5,\ A_2$	Kosten:	4
adda.L	$\#-8,\ A_2$	Kosten:	12
adda.L	$D_1,\ A_2$	Kosten:	8
move	$(A_2),\ D_2$	Kosten:	8
addq	$\#2,\ D_2$	Kosten:	4
move.L	$A_5,\ A_3$	Kosten:	4
adda.L	$\#-4,\ A_3$	Kosten:	12
move	$D_2,\ (A_3)$	Kosten:	8

Gesamtkosten : 124

Diskussion:

→ Die Folge ohne komplexe Adressierungsarten ist erheblich teurer :-(

→ Sie benötigt auch mehr Hilfsregister :-(

→ Die beiden Folgen sind nur äquivalent im Hinblick auf den Speicher — die Register haben anschließend verschiedene Inhalte ...

→ Eine korrekte Folge von Instruktionen kann als eine Pflasterung des Syntaxbaums aufgefasst werden !!!

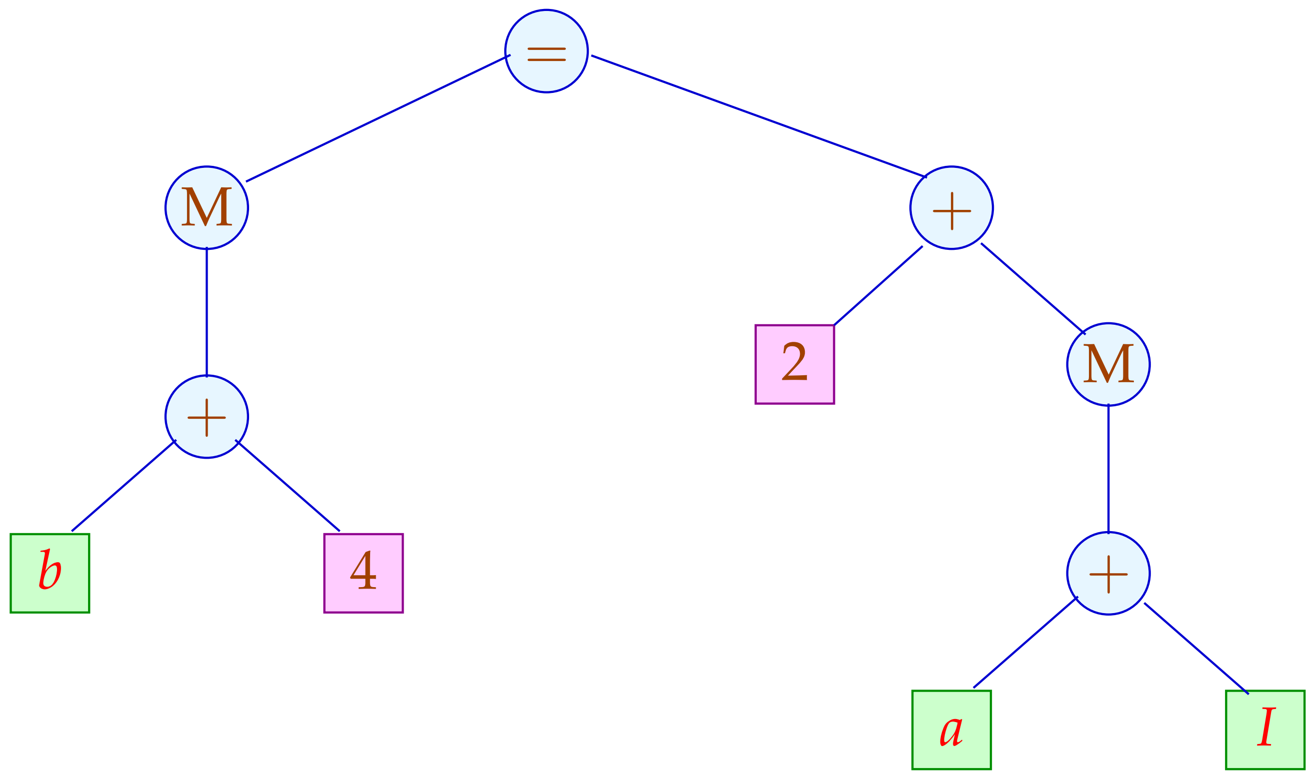
Genereller Ansatz:

- Wir betrachten Basis-Blöcke vor der Registerverteilung:

$$
\begin{aligned}
A &= a + I;\\
D_1 &= M[A];\\
D_2 &= D_1 + 2;\\
B &= b + 4;\\
M[B] &= D_2
\end{aligned}
$$

- Wir fassen diese als Folge von Bäumen auf. Wurzeln:
  - Werte, die mehrmals verwendet werden;
  - Variablen, die am Ende des Blocks lebendig sind;
  - Stores.

... im Beispiel:



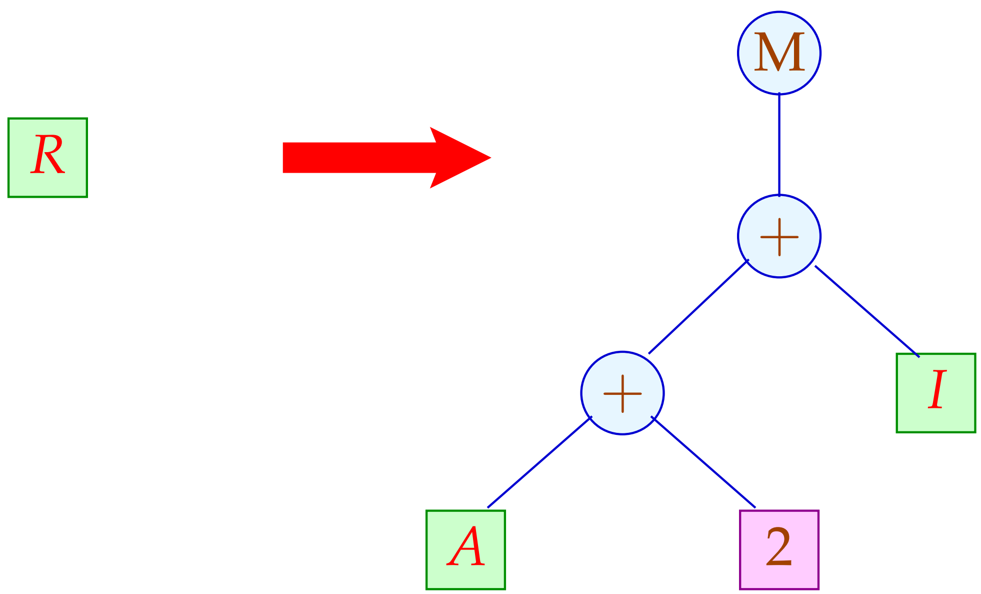
Die Hilfsvariablen A, B, D_1, D_2 sind vorerst verschwunden :-)

Idee:

Beschreibe den Effekt einer Instruktion als Ersetzungsregel auf Bäumen:

Die Instruktion: $R = M[A + 2 + D];$

entspricht zum Beispiel:

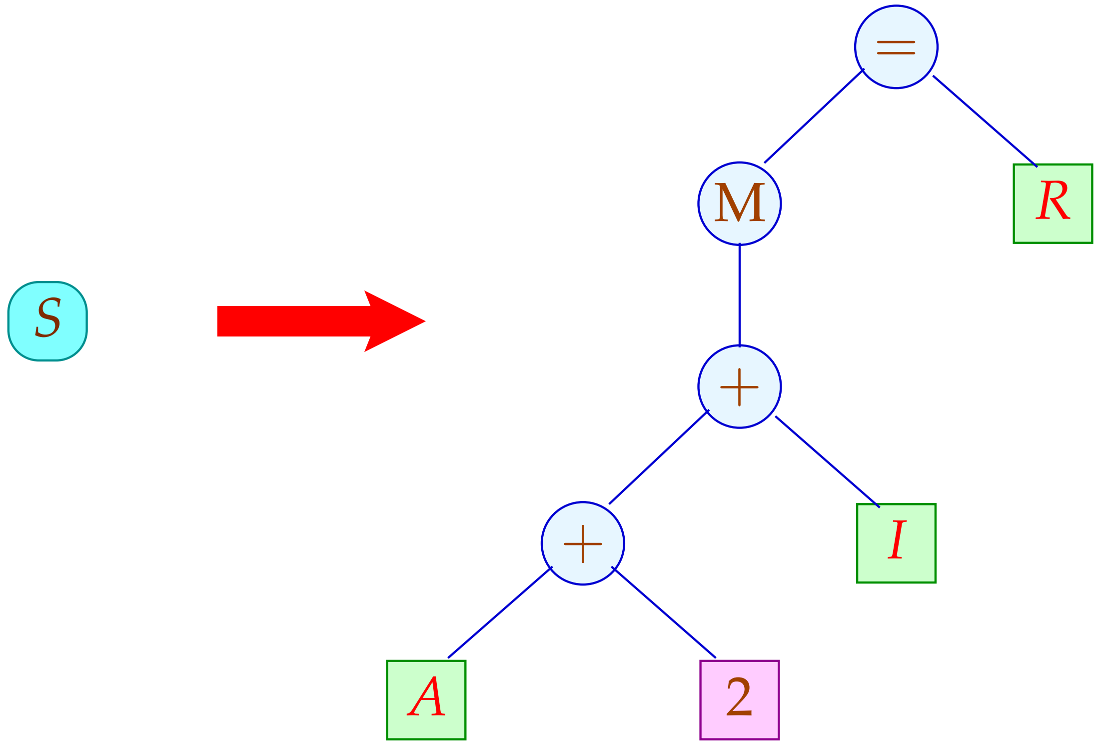


linke Seite	Ergebnisregister(klasse)
rechte Seite	berechneter Wert für Ergebnisregister
innere Knoten	• Load M • Arithmetik
Blätter	• Argumentregister(klassen) • Konstanten(klasse)

Die Grundidee erweitern wir (evt.) um eine Store-Operation.

Für die Instruktion: $M[A + 2 + D] = R;$

erlauben wir uns:



Die linke Seite S kommt nicht in rechten Seiten vor :-)

Spezifikation des Instruktionssatzes:

(1)	verfügbare Registerklassen	//	Nichtterminale
(2)	Operatoren und Konstantenklassen	//	Terminale
(3)	Instruktionen	//	Regeln

$\Longrightarrow$ reguläre Baumgrammatik

Triviales Beispiel:

Loads :	Comps :	Moves :
$D \to M[A]$	$D \to c$	$D \to A$
$D \to M[A + A]$	$D \to D + D$	$A \to D$

- Registerklassen D (Data) und A (Address).
- Arithmetik wird nur für Daten unterstützt ...
- Laden nur für Adressen :-)
- Zwischen Daten- und Adressregistern gibt es Moves.

Target: $M[A + c]$

Aufgabe:

Finde Folge von Regelanwendungen, die das Target aus einem Nichtterminal erzeugt ...

$$\boxed{\boxed{D}}$$

Target: $M[A + c]$

Aufgabe:

Finde Folge von Regelanwendungen, die das Target aus einem Nichtterminal erzeugt ...

Target: $M[A + c]$

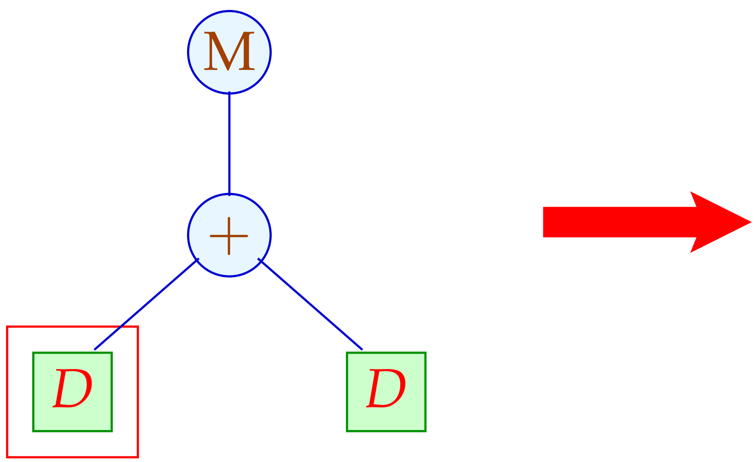
Aufgabe:

Finde Folge von Regelanwendungen, die das Target aus einem Nichtterminal erzeugt ...

Target: $M[A + c]$

Aufgabe:

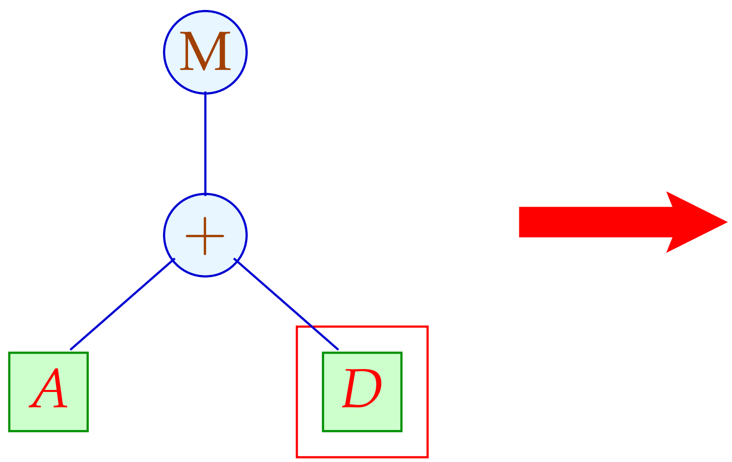
Finde Folge von Regelanwendungen, die das Target aus einem Nichtterminal erzeugt ...



Target: $M[A + c]$

Aufgabe:

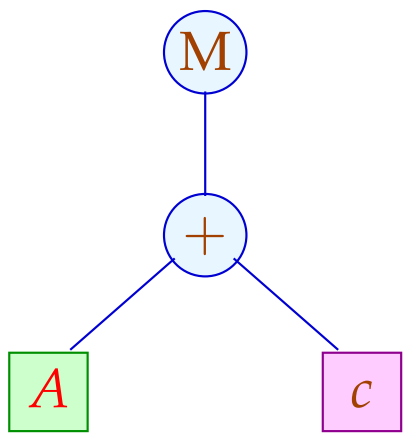
Finde Folge von Regelanwendungen, die das Target aus einem Nichtterminal erzeugt ...



Target: $M[A + c]$

Aufgabe:

Finde Folge von Regelanwendungen, die das Target aus einem Nichtterminal erzeugt ...



Die umgekehrte Folge der Regelanwendungen liefert eine geeignete Instruktionsfolge :-)

Verschiedene Ableitungen liefern verschiedene Folgen ...

Problem:

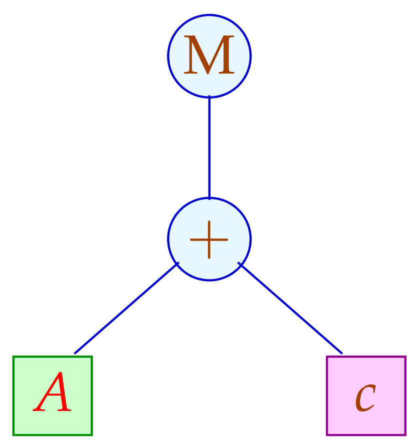
→ Wie durchsuchen wir systematisch die Menge aller Ableitungen ?

→ Wie finden wir die beste ??

Beobachtung:

- Nichtterminale stehen stets an den Blättern.
- Statt eine Ableitung für das Target topdown zu raten, sammeln wir sämtliche Möglichkeiten bottom-up auf

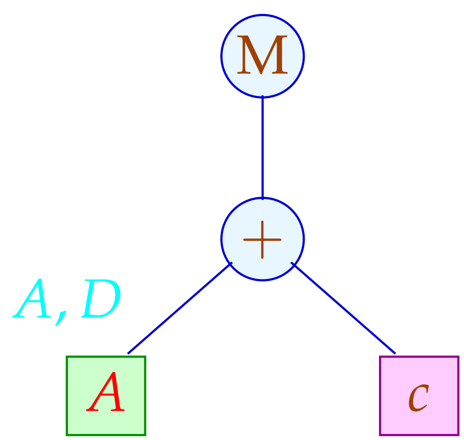
  $\Longrightarrow$ Tree parsing
- Dazu lesen wir die Regeln von rechts nach links ...



Beobachtung:

- Nichtterminale stehen stets an den Blättern.
- Statt eine Ableitung für das Target topdown zu raten, sammeln wir sämtliche Möglichkeiten bottom-up auf

  $\Longrightarrow$ Tree parsing
- Dazu lesen wir die Regeln von rechts nach links ...

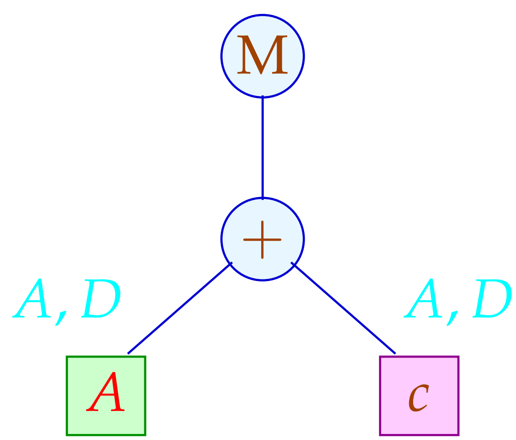


Beobachtung:

- Nichtterminale stehen stets an den Blättern.
- Statt eine Ableitung für das Target topdown zu raten, sammeln wir sämtliche Möglichkeiten bottom-up auf

$\Longrightarrow$ Tree parsing

- Dazu lesen wir die Regeln von rechts nach links ...

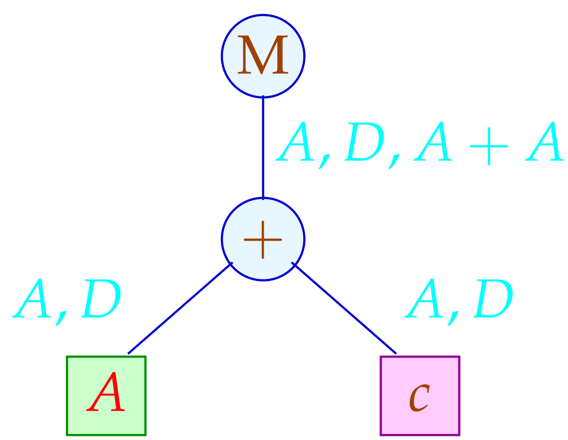


Beobachtung:

- Nichtterminale stehen stets an den Blättern.
- Statt eine Ableitung für das Target topdown zu raten, sammeln wir sämtliche Möglichkeiten bottom-up auf

$\Longrightarrow$ Tree parsing

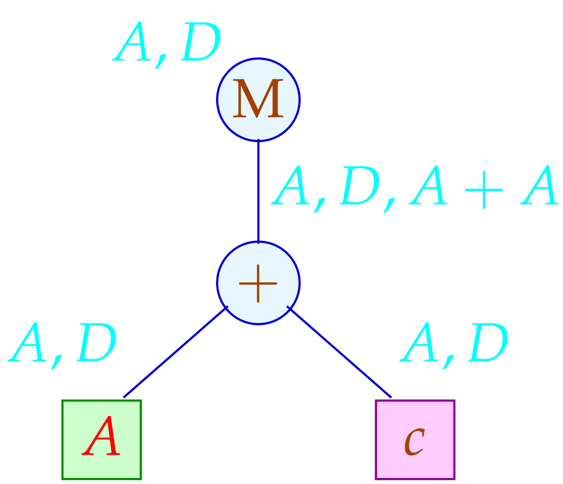
- Dazu lesen wir die Regeln von rechts nach links ...



Beobachtung:

- Nichtterminale stehen stets an den Blättern.
- Statt eine Ableitung für das Target topdown zu raten, sammeln wir sämtliche Möglichkeiten bottom-up auf

  $\Longrightarrow$ Tree parsing
- Dazu lesen wir die Regeln von rechts nach links ...



Für jeden Teilbaum t des Targets sammeln wir die Menge

$$Q(t) \subseteq \{S\} \cup \mathsf{Reg} \cup \mathsf{Term}$$

Reg die Menge der Registerklassen,
Term die Menge der Teilbäume rechter Seiten — auf mit:

$$Q(t) = \{s \mid s \Rightarrow^* t\}$$

Diese ergeben sich zu:

$$\begin{array}{lll} Q(R) & = & \mathsf{Move}\,\{R\} \\ Q(c) & = & \mathsf{Move}\,\{c\} \\ Q(a(t_1,\ldots,t_k)) & = & \mathsf{Move}\,\{s = a(s_1,\ldots,s_k) \in \mathsf{Term} \mid s_i \in Q(t_i)\} \end{array}$$

// normalerweise $k \leq 2$:-)

Die Hilfsfunktion **Move** bildet den Abschluss unter Regelanwendungen:

$$\textsf{Move}\,(L) \quad \supseteq \quad L$$

$$\textsf{Move}\,(L) \quad \supseteq \quad \{R \in \textsf{Reg} \mid \exists\, s \in L : \quad R \to s\}$$

Die kleinste Lösung dieses Constraint-Systems lässt sich aus der Grammatik in linearer Zeit berechnen :-)

// Im Beispiel haben wir in $Q(t)$ auf s verzichtet,
// falls s kein echter Teilterm einer rechten Seite ist :-)

Auswahlkriterien:

- Länge des Codes;
- Laufzeit der Ausführung;
- Parallelisierbarkeit;
- ...

Achtung:

Die Laufzeit von Instruktionen kann vom Kontext abhängen !!?

Vereinfachung:

Jede Instruktion r habe Kosten $c[r]$.

Die Kosten einer Instruktionsfolge sind additiv:

$$c[r_1 \ldots r_k] \;=\; c[r_1] + \ldots + c[r_k]$$

	c	Instruktion
0	3	$D \rightarrow M[A+A]$
1	2	$D \rightarrow M[A]$
2	1	$D \rightarrow D+D$
3	1	$D \rightarrow c$
4	1	$D \rightarrow A$
5	1	$A \rightarrow D$

Aufgabe:

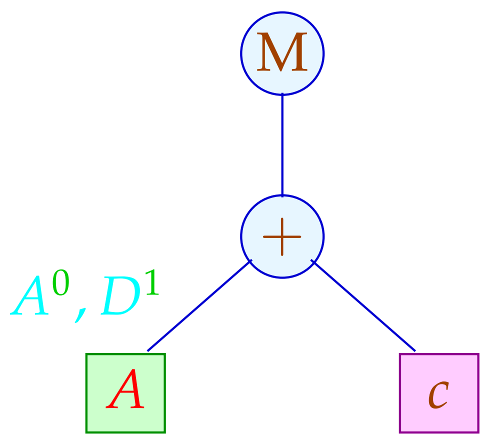
Wähle eine Instruktionsfolge mit minimalen Kosten !

Idee:

Sammle Ableitungen bottom-up auf unter

* Kostenkalkulation und
* Auswahl.

... im Beispiel:

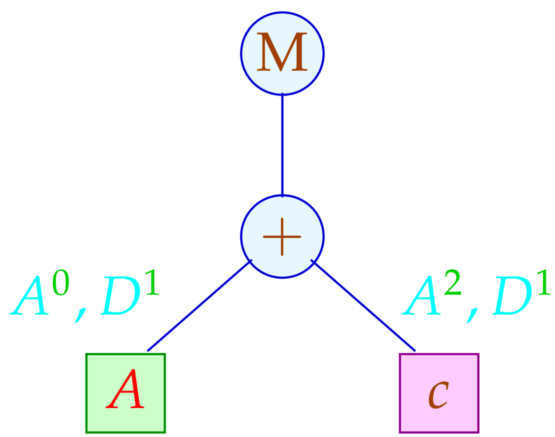


Idee:

Sammle Ableitungen bottom-up auf unter

* Kostenkalkulation und
* Auswahl.

... im Beispiel:

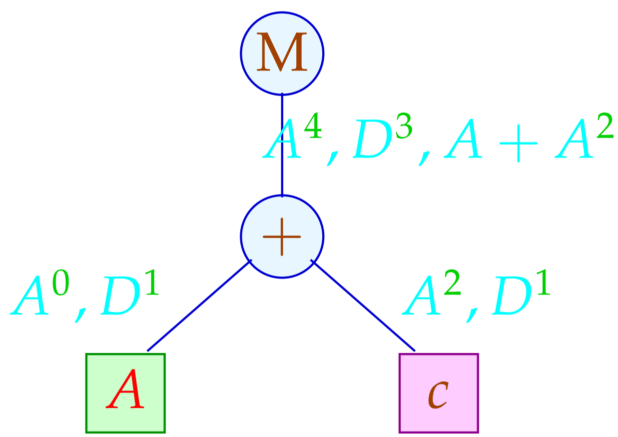


Idee:

Sammle Ableitungen bottom-up auf unter

* Kostenkalkulation und
* Auswahl.

... im Beispiel:

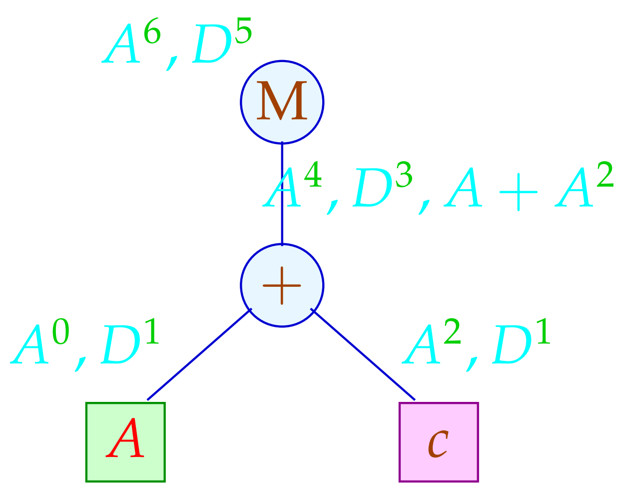


Idee:

Sammle Ableitungen bottom-up auf unter

* Kostenkalkulation und
* Auswahl.

... im Beispiel:

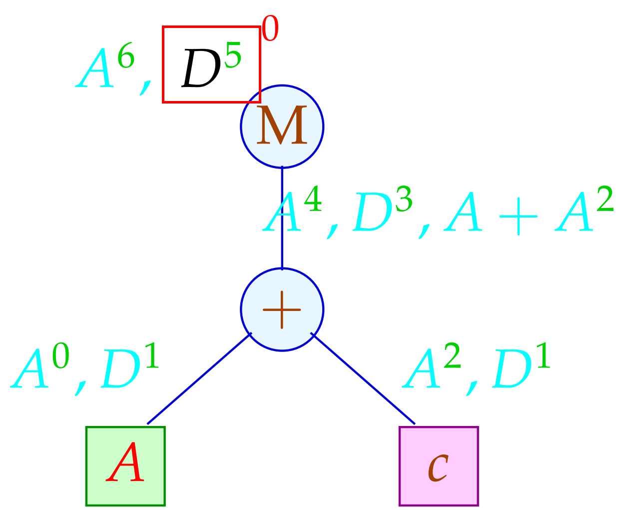


Idee:

Sammle Ableitungen bottom-up auf unter

- Kostenkalkulation und
- Auswahl.

... im Beispiel:

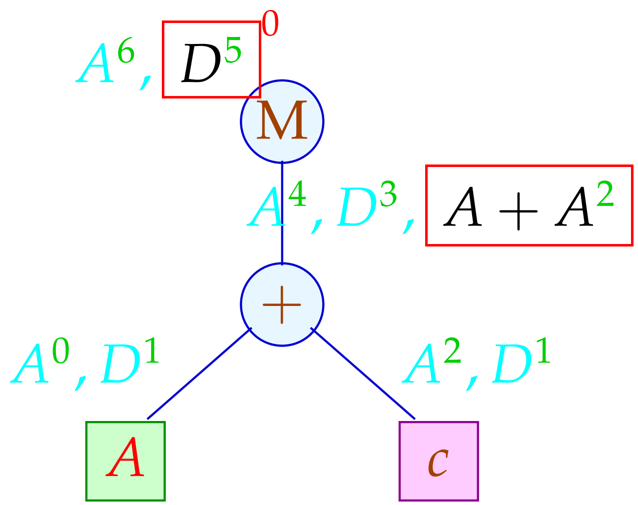


Idee:

Sammle Ableitungen bottom-up auf unter

- Kostenkalkulation und
- Auswahl.

... im Beispiel:

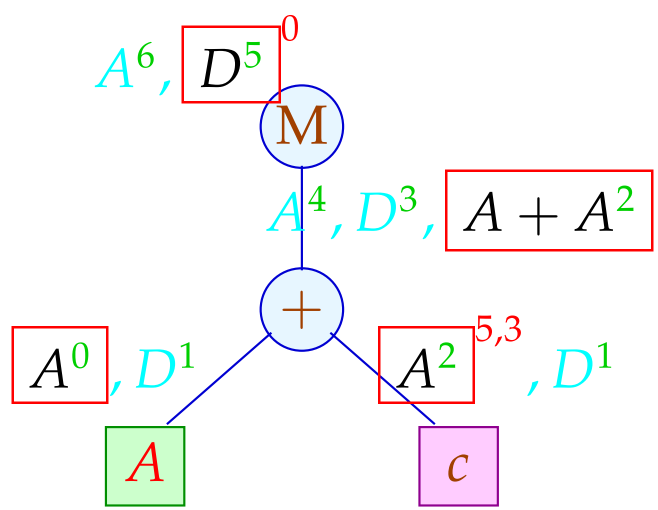


Idee:

Sammle Ableitungen bottom-up auf unter

* Kostenkalkulation und
* Auswahl.

... im Beispiel:



Kostenkalkulation:

$$
\begin{array}{lcll}
c_t[s] & = & c_{t_1}[s_1] + \ldots + c_{t_k}[s_k] & \text{falls} \quad s = a(s_1,\ldots,s_k), t = a(t_1,\ldots,t_k) \\
c_t[R] & = & \bigsqcap\{c[R,s] + c_t[s] \mid s \in Q(t)\} & \text{wobei}
\end{array}
$$

$$
\begin{array}{lcll}
c[R,s] & \leq & c[r] & \text{falls} \quad r \; : \; R \rightarrow s \\
c[R,s] & \leq & c[r] + c[R',s] & \text{falls} \quad r \; : \; R \rightarrow R'
\end{array}
$$

Das Constraint-System für $c[R,s]$ kann in Zeit $\mathcal{O}(n \cdot \log n)$ gelöst werden — falls n die Anzahl der Paare R, s ist :-)

Für jedes R, s liefert die Fixpunkt-Berechnung eine Folge:

$$
\pi[R,s] \; : \; R \Rightarrow R_1 \Rightarrow \ldots \Rightarrow R_k \Rightarrow s
$$

deren Kosten gerade $c[R,s]$ ist :-)

Mithilfe der $\pi[R, s]$ lässt sich eine billigste Ableitung topdown rekonstruieren :-)

Im Beispiel:

$$\begin{array}{lll} D_2 & = & c; \\ A_2 & = & D_2; \\ D_1 & = & M[A_1 + A_2]; \end{array}$$

mit Kosten 5 . Die Alternative:

$$\begin{array}{lll} D_2 & = & c; \\ D_3 & = & A_1; \\ D_4 & = & D_3 + D_2; \\ A_2 & = & D_4; \\ D_1 & = & M[A_2]; \end{array}$$

hätte Kosten 7 :-)

Diskussion:

- Die Code-Erzeugung muss schnell gehn :-)
- Anstelle für jeden Knoten neu zu überprüfen, wie die Regeln zusammen passen, kann die Berechnung auch in einen endlichen Automaten kompiliert werden :-))

Ein deterministischer endlicher Baumautomat (DTA) A besteht aus:

Q	$==$	endliche Menge von Zuständen
Σ	$==$	Operatoren und Konstanten
δ_a	$==$	Übergangsfunktion für $a \in \Sigma$
$F \subseteq Q$	$==$	akzeptierende Zustände

Dabei ist:

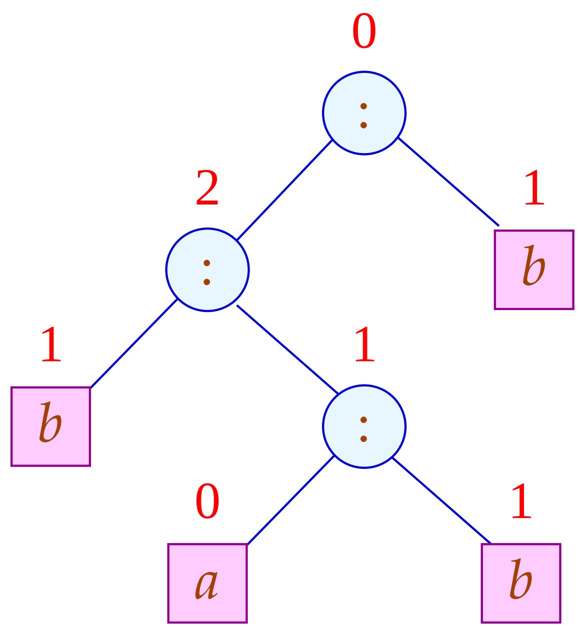
$$\begin{array}{llll} \delta_c : & Q & \text{falls} & c \quad \text{Konstante} \\ \delta_a : & Q^k \to Q & \text{falls} & a \quad k\text{-stellig} \end{array}$$

Beispiel:

$$\begin{array}{lllllll} Q & = & \{0,1,2\} & & F & = & \{0\} \\ \Sigma & = & \{a,b,:\} & & & & \\ \delta_a & = & 0 & & \delta_b & = & 1 \end{array}$$

$$\delta_:(s_1,s_2) = (s_1+s_2)\%3$$

// akzeptiert alle Bäume mit $3 \cdot k$ b-Blättern



Der Zustand an einem Knoten a ergibt sich aus den Zuständen der Kinder mittels δ_a (-:

$$
\begin{array}{lll}
Q(c) & = & \delta_c \\
Q(a(t_1, \ldots, t_k)) & = & \delta_a(Q(t_1), \ldots, Q(t_k))
\end{array}
$$

Die von A definierte Sprache (oder: Menge von Bäumen) ist:

$$\mathcal{L}(A) = \{t \mid Q(t) \in F\}$$

... in unserer Anwendung:

$$Q \quad == \quad \text{Teilmengen von } \mathsf{Reg} \cup \mathsf{Term} \cup \{S\}$$

// I.a. werden nicht sämtiche Teilmengen benötigt :-)

$$F \quad == \quad \text{gewünschter Effekt}$$

$$\delta_R \quad == \quad \mathsf{Move}\,\{R\}$$

$$\delta_c \quad == \quad \mathsf{Move}\,\{c\}$$

$$\delta_a(Q_1, \ldots, Q_k) \quad == \quad \mathsf{Move}\,\{s = a(s_1, \ldots, s_k) \in \mathsf{Term} \mid s_i \in Q_i\}$$

... im Beispiel:

$$\begin{array}{rcl} \delta_c & = & \{A, D\} \quad = \quad q_0 \\ & = & \delta_A \\ & = & \delta_D \end{array}$$

$$\begin{array}{rcl} \delta_+(q_0, q_0) & = & \{A, D, A + A\} \quad = \quad q_1 \\ & = & \delta_+(q_0, \_) \\ & = & \delta_+(\_, q_0) \end{array}$$

$$\begin{array}{rcl} \delta_M(q_0) & = & \{A, D\} \quad = \quad q_0 \\ & = & \delta_M(q_1) \end{array}$$

Um die Anzahl der Zustände zu reduzieren, haben wir die vollständigen rechten Seiten, die keine echten Teilmuster sind, in den Zuständen weggelassen :-)

Integration der Kostenberechnung:

Problem:

Kosten können (im Prinzip) beliebig groß werden ;-(

Unser FTA besitzt aber nur endlich viele Zustände :-((

Idee:

Pelegri-Lopart 1988

Betrachte nicht absolute Kosten — sondern relative !!!

Eduardo Pelegri-Llopart,
Sun Microsystems, Inc.

Beobachtung:

- In gängigen Prozessoren kann man Werte von jedem Register in jedes andere schieben $\Longrightarrow$

  Die Kosten zwischen Registern differieren nur um eine Konstante :-)

- Komplexe rechte Seiten lassen sich i.a. mittels elementarerer Instruktionen simulieren $\Longrightarrow$

  Die Kosten zwischen Teilausdrücken und Registern differieren nur um eine Konstante :-))

- Die Kostenberechnung ist additiv $\Longrightarrow$

  Wir können statt mit absoluten Kosten-Angaben auch mit Kosten-Differenzen rechnen !!!

  Von diesen gibt es nur endlich viele :-)

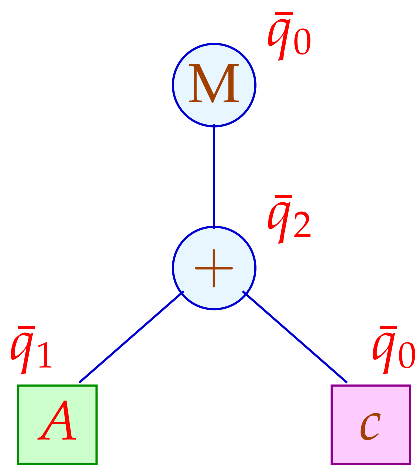
... im Beispiel:

$$
\begin{array}{rcl}
\delta_c & = & \{A \mapsto 1, D \mapsto 0\} \quad = \quad \bar{q}_0 \\
 & = & \delta_D \\
\delta_A & = & \{A \mapsto 0, D \mapsto 1\} \quad = \quad \bar{q}_1
\end{array}
$$

$$
\begin{array}{rcl}
\delta_+(\bar{q}_1, \bar{q}_0) & = & \{A \mapsto 2, D \mapsto 1, A + A \mapsto 0\} \quad = \quad \bar{q}_2 \\
\delta_+(\bar{q}_0, \bar{q}_0) & = & \{A \mapsto 1, D \mapsto 0, A + A \mapsto 1\} \quad = \quad \bar{q}_3 \\
\delta_+(\bar{q}_1, \bar{q}_1) & = & \{A \mapsto 4, D \mapsto 3, A + A \mapsto 0\} \quad = \quad \bar{q}_4 \\
 & \ldots &
\end{array}
$$

$$
\begin{array}{rcl}
\delta_M(\bar{q}_2) & = & \{A \mapsto 1, D \mapsto 0\} \quad = \quad \bar{q}_0 \\
 & = & \delta_M(\bar{q}_i) \qquad , \quad i = 0, \ldots, 4
\end{array}
$$

... das liefert die folgende Berechnung:



Für jede Konstanten-Klasse c und jedes Register R in δ_c tabellieren wir die zu wählende billigste Berechnung:

$$c \; : \quad \{A \mapsto 5, 3, D \mapsto 3\}$$

Analog tabellieren wir für jeden Operator a , jedes $\tau \in \bar{Q}^k$ und jedes R in $\delta_a(\tau)$:

M	$\textsf{select}_M$
$\bar{q}_0$	$\{A \mapsto 5, 1, D \mapsto 1\}$
$\bar{q}_1$	$\{A \mapsto 5, 1, D \mapsto 1\}$
$\bar{q}_2$	$\{A \mapsto 5, 0, D \mapsto 0\}$
$\bar{q}_3$	$\{A \mapsto 5, 1, D \mapsto 1\}$
$\bar{q}_4$	$\{A \mapsto 5, 0, D \mapsto 0\}$

Für "$+$" ist die Tabelle besonders einfach:

$+$	$\bar{q}_j$
$\bar{q}_i$	$\{A \mapsto 5, 3, D \mapsto 3\}$

Problem:

- Für reale Instruktionssätze benötigt man leicht um die 1000 Zustände.
- Die Tabellen für mehrstellige Operatoren werden riesig :-(

$\Longrightarrow$ Wir benötigen Verfahren der Tabellen-Komprimierung ...

Tabellen-Kompression:

Die Tabelle für "$+$" sieht im Beispiel so aus:

$+$	$\bar{q}_0$	$\bar{q}_1$	$\bar{q}_2$	$\bar{q}_3$	$\bar{q}_4$
$\bar{q}_0$	$\bar{q}_3$	$\bar{q}_2$	$\bar{q}_3$	$\bar{q}_3$	$\bar{q}_3$
$\bar{q}_1$	$\bar{q}_2$	$\bar{q}_4$	$\bar{q}_2$	$\bar{q}_2$	$\bar{q}_2$
$\bar{q}_2$	$\bar{q}_3$	$\bar{q}_2$	$\bar{q}_3$	$\bar{q}_3$	$\bar{q}_3$
$\bar{q}_3$	$\bar{q}_3$	$\bar{q}_2$	$\bar{q}_3$	$\bar{q}_3$	$\bar{q}_3$
$\bar{q}_4$	$\bar{q}_3$	$\bar{q}_2$	$\bar{q}_3$	$\bar{q}_3$	$\bar{q}_3$

Die meisten Zeilen / Spalten sind offenbar ganz ähnlich ;-)

Idee 1: Äquivalenzklassen

Wir setzen $q \equiv_a q'$, genau dann wenn

$$\forall\, p : \quad \delta_a(q,p) = \delta_a(q',p) \quad \wedge \quad \delta_a(p,q) = \delta_a(p,q')$$
$$\wedge \quad \mathsf{select}_a(q,p) = \mathsf{select}_a(q',p) \quad \wedge \quad \mathsf{select}_a(p,q) = \mathsf{select}_a(p,q')$$

Im Beispiel:

$$\begin{array}{lll} Q_1 & = & \{\bar{q}_0, \bar{q}_2, \bar{q}_3, \bar{q}_4\} \\ Q_2 & = & \{\bar{q}_1\} \end{array}$$

mit:

$+$	Q_1	Q_2
Q_1	$\bar{q}_3$	$\bar{q}_2$
Q_2	$\bar{q}_2$	$\bar{q}_4$

Idee 2: Zeilenverschiebung

Sind viele Einträge gleich (im Beispiel etwa default $= \bar{q}_3$), genügt es, die übrigen Einträge zu speichern ;-)

Im Beispiel:

$+$	$\bar{q}_0$	$\bar{q}_1$	$\bar{q}_2$	$\bar{q}_3$	$\bar{q}_4$
$\bar{q}_0$		$\bar{q}_2$			
$\bar{q}_1$	$\bar{q}_2$	$\bar{q}_4$	$\bar{q}_2$	$\bar{q}_2$	$\bar{q}_2$
$\bar{q}_2$		$\bar{q}_2$			
$\bar{q}_3$		$\bar{q}_2$			
$\bar{q}_4$		$\bar{q}_2$			

Dann legen wir:

(1) gleiche Zeilen übereinander;

(2) verschiedene (Klassen von) Zeilen auf Lücke verschoben übereinander:

	$\bar{q}_0$	$\bar{q}_1$	$\bar{q}_2$	$\bar{q}_3$	$\bar{q}_4$
class	0	1	0	0	0

	0	1
disp	0	2

	0	1	2	3	4	5	6
A	$\bar{q}_2$	$\bar{q}_2$	$\bar{q}_4$	$\bar{q}_2$	$\bar{q}_2$	$\bar{q}_2$	$\bar{q}_2$
valid	0	0	1	1	1	1	1

Für jeden Eintrag im ein-dimensionalen Feld A vermerken wir in valid , zu welcher Zeile der Eintrag gehört ...

Ein Feld-Zugriff $\delta_+(\bar{q}_i, \bar{q}_j)$ wird dann so realisiert:

$$
\begin{array}{lll}
\delta_+(\bar{q}_i, \bar{q}_j) & = & \mathsf{let} \quad c \;=\; \mathsf{class}[\bar{q}_i]; \\
& & \qquad\;\; d \;=\; \mathsf{disp}[c]; \\
& & \mathsf{in} \;\; \mathsf{if} \;\; (\mathsf{valid}[d+j] \equiv c) \\
& & \qquad \mathsf{then} \;\; \mathsf{A}[d+j] \\
& & \qquad \mathsf{else} \;\; \mathsf{default} \\
& & \mathsf{end}
\end{array}
$$

Reinhard Wilhelm, Saarbrücken

Diskussion:

- Die Tabellen werden i.a. erheblich kleiner.
- Dafür werden Tabellenzugriffe etwas teurer.
- Das Verfahren versagt in einigen (theoretischen) Fällen.
- Dann bleibt immer noch das dynamische Verfahren ...

  möglicherweise mit Caching der einmal berechneten Werte, um unnötige Mehrfachberechnungen zu vermeiden :-)

3.3 Instruction Level Parallelität

Moderne Prozessoren führen nicht eine Instruktion nach der anderen aus.

Wir betrachten hier zwei Ansätze:

(1) VLIW (Very Large Instruction Words)
(2) Pipelining

VLIW:

Eine Instruktion führt simultan bis zu k (etwa 4:-) elementare Instruktionen aus.

Pipelining:

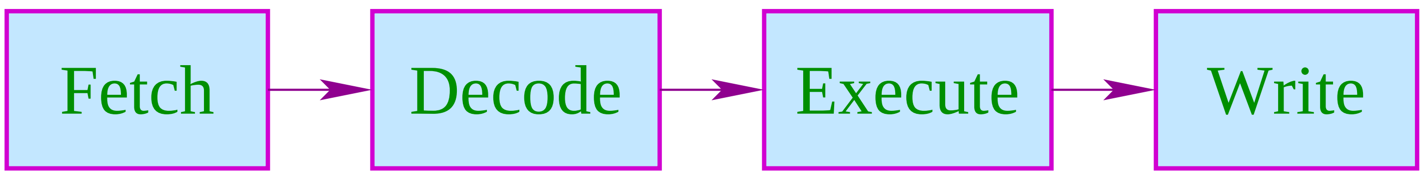
Instruktionsausführungen können zeitlich überlappen.

Beispiel:

$$w = (R_1 = R_2 + R_3 \mid D = D_1 * D_2 \mid R_3 = M[R_4])$$

Achtung:

- Instruktionen belegen Hardware-Einrichtungen.
- Instruktionen greifen auf die gleichen Register zu $\Longrightarrow$ Hazards
- Ergebnisse einer Insgtruktion liegen erst nach einiger Zeit vor.
- Während dieser Zeit wechselt i.a. die benutzte Hardware:



- Während Execute bzw. Write werden evt. unterschiedliche interne Register/Busse/Alus benutzt.

Wir schließen:

Aufteilung der Instruktionsfolge in Wörter und ihre Aufeinanderfolge ist Restriktionen unterworfen ...

Im folgenden ignorieren wir die Phasen Fetch und Decode :-)

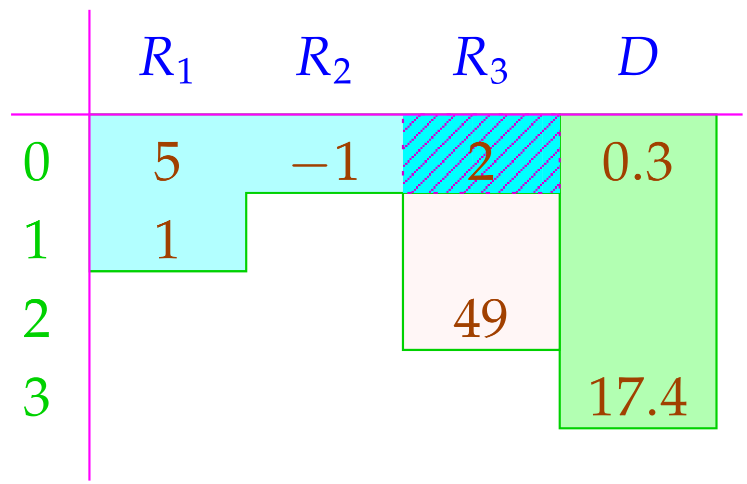
Beispiele für Restriktionen:

(1) maximal ein Load/Store pro Wort;
(2) maximal ein Jump;
(3) maximal ein Write in das selbe Register.

Timing:

Gleitkomma-Operation	3
Laden/Speichern	2
Integer-Arithmetik	1

Timing-Diagramm:



R_3 wird überschrieben, nachdem die Addition 2 abgeholte :-)

Wird auf ein Register mehrfach zugegriffen (hier: R_3), wird eine Strategie zur Konfliktlösung benötigt ...

Konflikte:

Read-Read: Ein Register wird mehrfach ausgelesen.

$\Longrightarrow$ i.a. unproblematisch :-)

Read-Write: Ein Register wird in einer Instruktion sowohl gelesen wie geschrieben.

Lösungsmöglichkeiten:

- ... verbieten!
- Lesen wird verzögert (stalls), bis Schreiben beendet ist!
- Lesen zeitlich vor dem Schreiben liefert den alten Wert!

  Gleichzeitiges Lesen wird verzögert/verboten/bevorzugt.

Write-Write: Ein Register wird mehrfach beschrieben.

$\Longrightarrow$ i.a. unproblematisch :-)

Lösungsmöglichkeiten:

- ... verbieten!
- ...

In unseren Beispielen ...

- erlauben wir gleichzeitiges Lesen;
- verbieten wir gleichzeitiges Schreiben bzw. Schreiben und Lesen;
- fügen wir keine Stalls ein.

Wir betrachten erst mal nur Basis-Blöcke, d.h. Folgen von Zuweisungen ...

Idee: Datenabhängigkeitsgraph

Knoten	Instruktionen
Kanten	Abhängigkeiten

Beispiel:

$$
\begin{array}{ll}
(1) & x = x + 1; \\
(2) & y = M[A]; \\
(3) & t = z; \\
(4) & z = M[A + x]; \\
(5) & t = y + z;
\end{array}
$$

Mögliche Abhängigkeiten:

Definition	$\longrightarrow$	Use	//	Reaching Definitions
Use	$\longrightarrow$	Definition	//	???
Definition	$\longrightarrow$	Definition	//	Reaching Definitions

Reaching Definitions: Ankommende Definitionen

Ermittle für jedes u , welche Variablen-Definitionen ankommen
$\Longrightarrow$ mithilfe Ungleichungssystem berechenbar :-)

Der abstrakte Bereich:

$\mathbb{R} = 2^{Nodes}$ // Man hätte auch Kanten nehmen können :-)

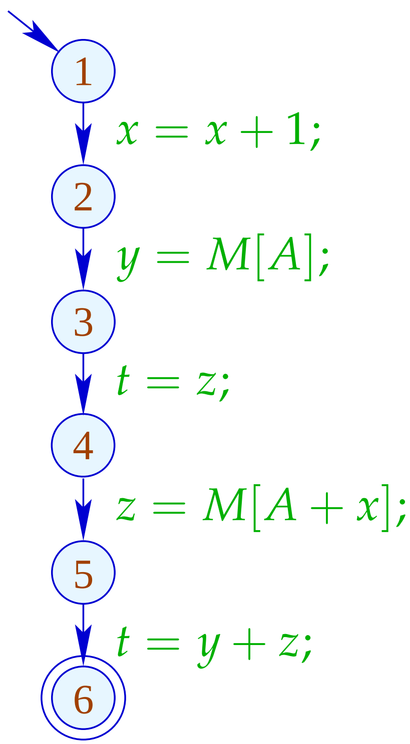
Die Transfer-Funktionen:

$$
\begin{array}{lll}
[\![(\_, ;, \_)]\!]^\sharp \, R & = & R \\
[\![(\_, Pos(e), \_)]\!]^\sharp \, R & = & [\![(\_, Neg(e), \_)]\!]^\sharp \, R \quad = \quad R \\
[\![(u, x = e;, \_)]\!]^\sharp \, R & = & (R \backslash Defs_x) \cup \{u\} \qquad \text{wobei} \\
 & & Defs_x \text{ die Menge der Definitionen von } x \text{ ist} \\
[\![(u, x = M[A];, \_)]\!]^\sharp \, R & = & (R \backslash Defs_x) \cup \{u\} \\
[\![(\_, M[A] = x;, \_)]\!]^\sharp \, R & = & R
\end{array}
$$

Die Information wird offenbar vorwärts propagiert, wobei die Ordnung auf dem vollständigen Verband $\mathbb{R}$ “$\subseteq$” ist :-)

Vor Programm-Ausführung ist die Menge der ankommenden Definitionen $d_0 = \{\bullet_x \mid x \in \mathit{Vars}\}$.

... im Beispiel:



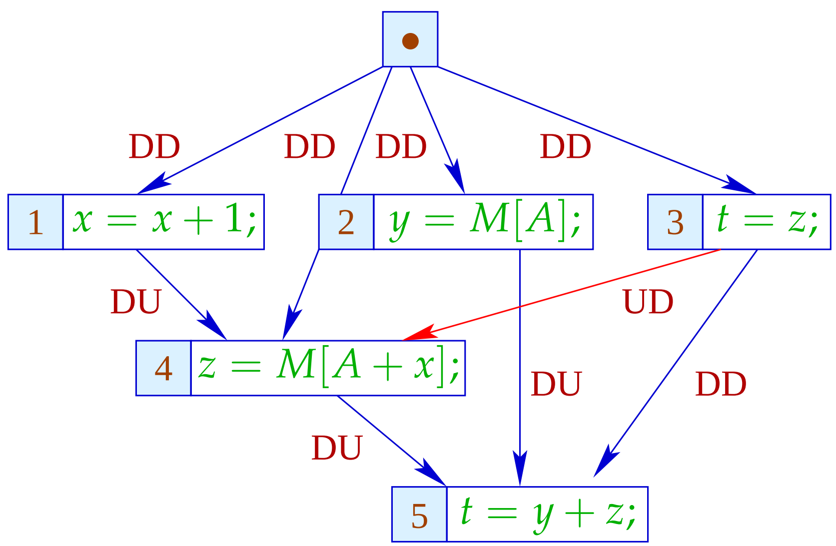
	$\mathcal{R}$
1	$\{\bullet_x, \bullet_y, \bullet_z, \bullet_t\}$
2	$\{1, \bullet_y, \bullet_z, \bullet_t\}$
3	$\{1{,}2, \bullet_z, \bullet_t\}$
4	$\{1{,}2{,}3, \bullet_z\}$
5	$\{1{,}2{,}3{,}4\}$
6	$\{1{,}2{,}4{,}5\}$

Seien U_i, D_i die Mengen der an einer von u_i ausgehenden Kante benutzten bzw. definierten Variablen. Dann gilt:

$$(u_1, u_2) \in DD \qquad \text{falls} \quad u_1 \in \mathcal{R}[u_2] \wedge D_1 \cap D_2 \neq \emptyset$$
$$(u_1, u_2) \in DU \qquad \text{falls} \quad u_1 \in \mathcal{R}[u_2] \wedge D_1 \cap U_2 \neq \emptyset$$

... im Beispiel:

		Def	*Use*
1	$x = x + 1;$	$\{x\}$	$\{x\}$
2	$y = M[A];$	$\{y\}$	$\{A\}$
3	$t = z;$	$\{t\}$	$\{z\}$
4	$z = M[A + x];$	$\{z\}$	$\{A, x\}$
5	$t = y + z;$	$\{t\}$	$\{y, z\}$



Die UD-Kante $(3, 4)$ haben wir eingefügt, um zu verhindern, dass z vor der Benutzung überschrieben wird :-)

Im nächsten Schritt versehen wir jede Instruktion mit (ihren benötigten Resourcen, insbesondere) ihrer Zeit.

Wir wollen eine möglichst parallele korrekte Wortfolge bestimmen.

Dazu verwalten wir den aktuellen System-Zustand:

$$\Sigma : \mathit{Vars} \to \mathbb{N}$$

$$\Sigma(x) \;\hat{=}\; \text{zu wartende Zeit, bis } x \text{ vorliegt}$$

Am Anfang:

$$\Sigma(x) = 0$$

Wir müssen als Invariante garantieren, dass alle Operationen bei Betreten des Basisblocks abgeschlossen sind :-)

Dann füllen wir sukzessive die Slots der Wort-Folge:

- Wir beginnen bei den minimalen Knoten des Abhängigkeitsgraphen.
- Können wir nicht alle Slots eines Worts füllen, fügen wir ; ein :-)
- Nach jeder eingefügten Instruktion berechnen wir Σ neu.

Achtung:

$\longrightarrow$ Die Ausführung zweier VLIWs kann überlappen !!!

$\longrightarrow$ Die Berechnung einer optimalen Folge ist NP-hart ...

Beispiel: Wortbreite $k = 2$

Wort		Zustand			
1	2	x	y	z	t
		0	0	0	0
$x = x + 1$	$y = M[A]$	0	1	0	0
$t = z$	$z = M[A + x]$	0	0	1	0
		0	0	0	0
$t = y + z$		0	0	0	0

In jedem Takt beginnt die Ausführung eines neuen Worts.

Im Zustand brauchen wir uns nur merken, wieviele Takte auf das Ergebnis noch gewartet werden muss :-)

Beachte:

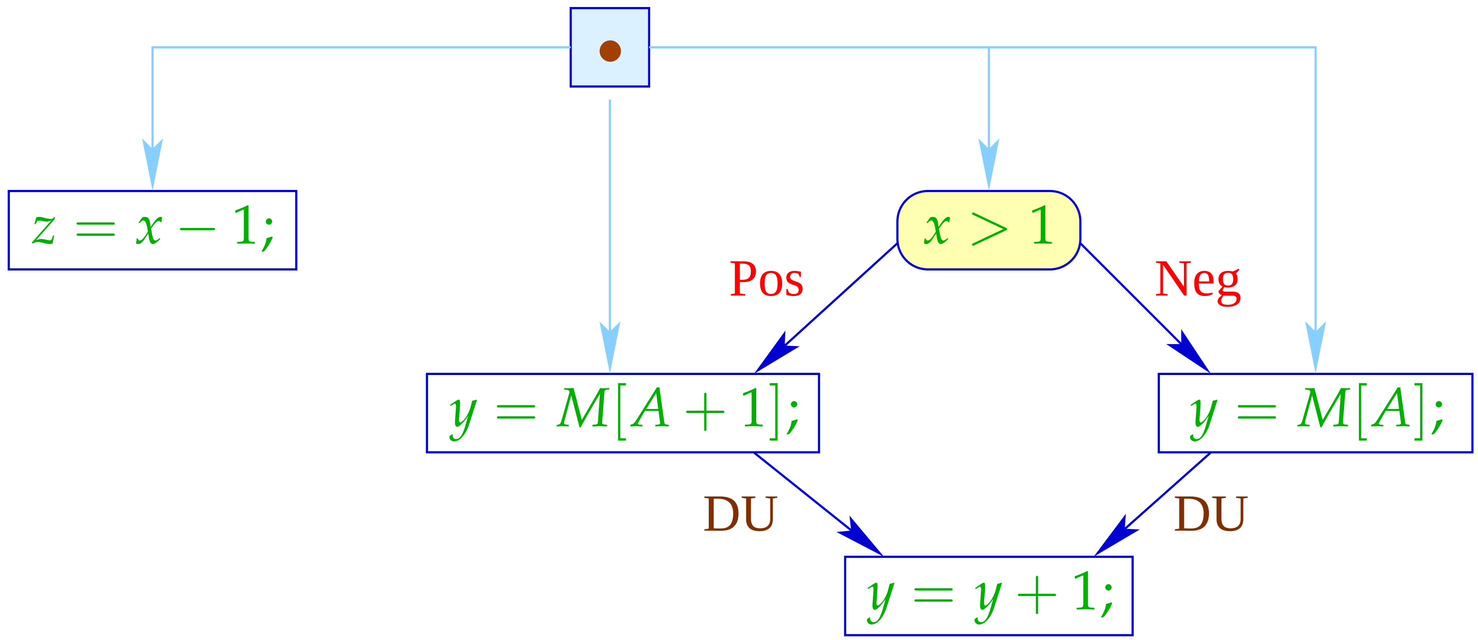
- Wenn Instruktionen zukünftiger Wortwahl weitere Restriktionen auferlegen, vermerken wir diese ebenfalls in Σ .
- Trotzdem unterscheiden wir nur endlich viele System-Zustände :-)
- Die Berechnung des Effekts eines VLIW auf Σ lässt sich in einen endlichen Automaten compilieren !!!
- Dieser Automat könnte allerdings sehr groß sein :-(
- Die Qual der billigsten Auswahl erspart er uns nicht :-(
- Basis-Blöcke sind leider i.a. nicht sehr groß

  $\Longrightarrow$ die Möglichkeiten zur Parallelisierung sind beschränkt :-((

Erweiterung 1: Azyklischer Code

$$
\begin{array}{l}
\mathsf{if}\ \ (x > 1)\ \ \{ \\
\qquad y = M[A]; \\
\qquad z = x - 1; \\
\}\ \ \mathsf{else}\ \ \{ \\
\qquad y = M[A+1]; \\
\qquad z = x - 1; \\
\} \\
y = y + 1;
\end{array}
$$

Im Abhängigkeitsgraph müssen wir zusätzlich die Kontroll-Abhängigkeiten vermerken ...



Das Statement $z = x - 1;$ wird mit immer den gleichen Argumenten in beiden Zweigen ausgeführt und modifiziert keine der sonst benutzten Variablen :-)

Wir hätten es ohnehin vor das `if` schieben können :-))

Als Code können wir deshalb erzeugen:

	$z = x - 1$	if $(!(x > 0))$ goto A
	$y = M[A]$	
	goto B	
$A :$	$y = M[A + 1]$	
$B :$	$y = y + 1$	

Bei jedem Einsprung garantieren wir die Invariante :-(

Erlauben wir mehrere (bekannte) Zustände beim Betreten eines Teil-Basisblocks, können wir für diesen Code erzeugen, der allen diesen Bedingungen entspricht.

... im Beispiel:

	$z = x - 1$	if $(!(x > 0))$ goto A
	$y = M[A]$	goto B
$A:$	$y = M[A+1]$	
$B:$		
	$y = y + 1$	

Reicht uns diese Parallelität immer noch nicht, könnten wir versuchen, spekulativ Arbeit vorziehen ...

Dazu erforderlich:

- eine Idee, welche Alternative häufiger gewählt wird;
- die falsche Ausführung darf zu keiner Katastrophe d.h. Laufzeitfehlern führen (z.B. wegen Division durch 0);
- die falsch Ausführung muss rückgängig gemacht werden können (evt. durch verzögertes Commit) oder darf keinen beobachtbaren Effekt haben ...

... im Beispiel:

	$z = x - 1$	$y = M[A]$	$\text{if } (x > 0) \text{ goto } B$
	$y = M[A+1]$		
$B:$			
	$y = y + 1$		

Im Fall $x \leq 0$ haben wir $y = M[A]$ zuviel ausgeführt.

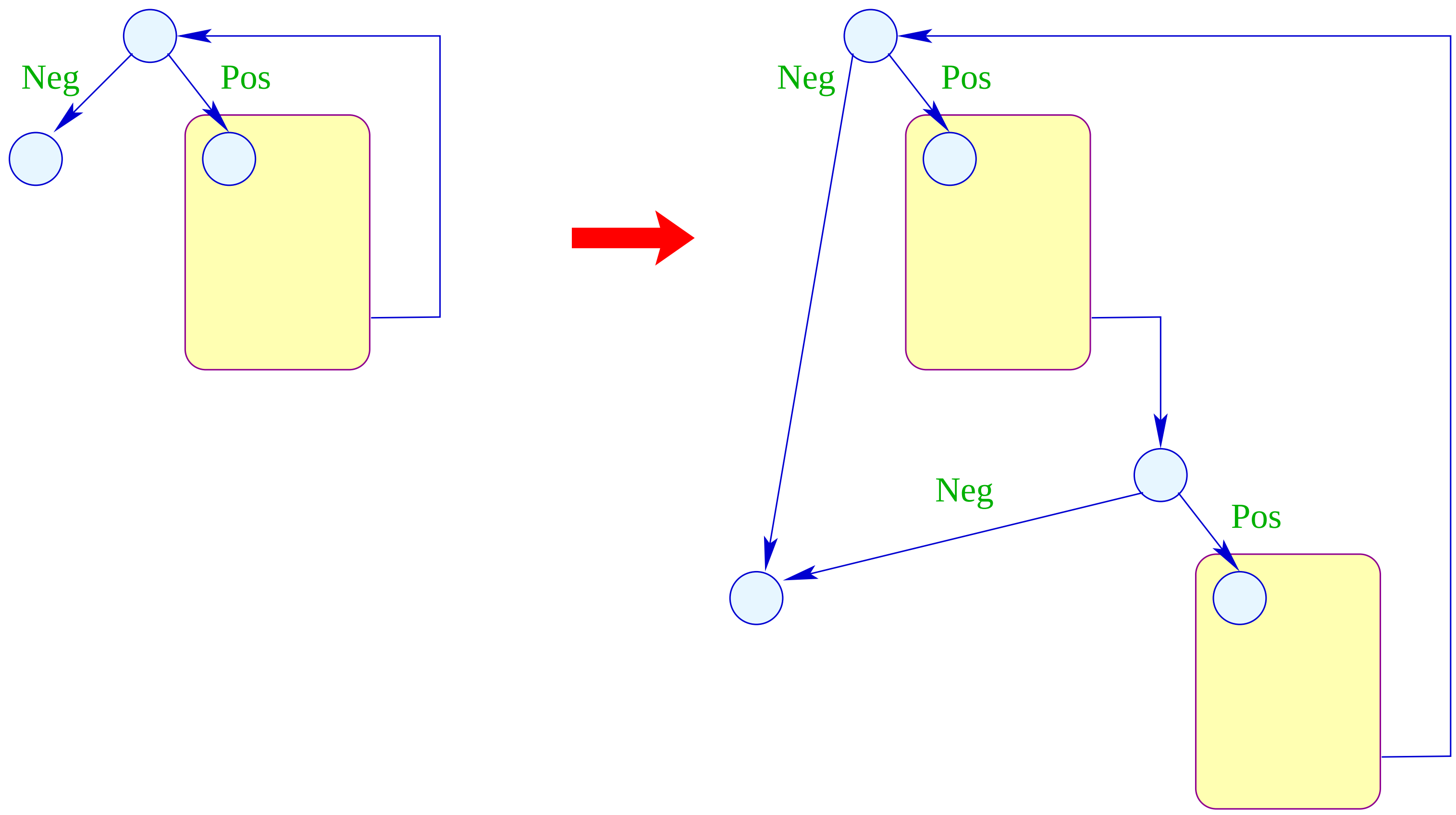
Dieser Wert wird aber im nächsten Schritt direkt überschrieben :-)

Allgemein:

$x = e;$ hat keinen beobachtbaren Effekt in einem Zweig, falls x in diesem Zweig tot ist :-)

Erweiterung 2: Abwickeln von Schleifen

Wir wickeln wichtige, d.h. innere Schleifen mehrmals ab:



Nun ist auch klar, welche Seite bei Tests zu begünstigen ist:

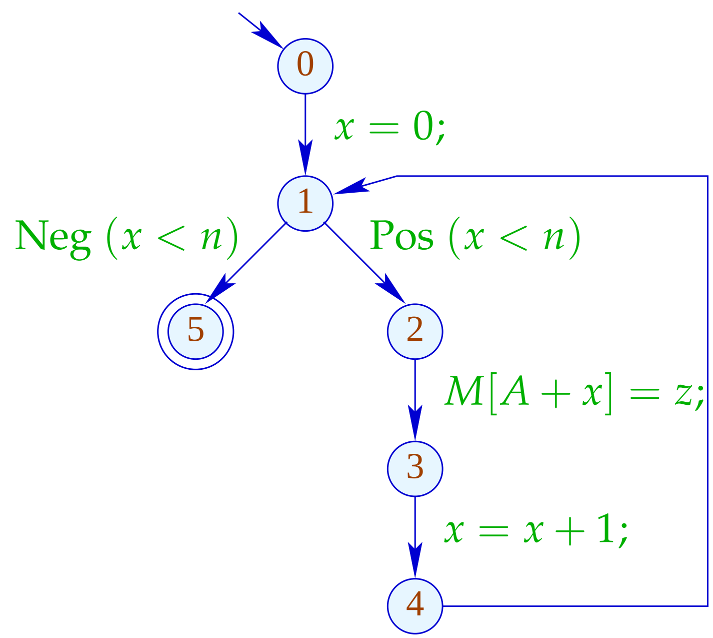
diejenige, die innerhalb des abgerollten Rumpfs der Schleife bleibt :-)

Achtung:

- Die verschiedenen Instanzen des Rumpfs werden relativ zu möglicherweise unterschiedlichen Anfangszuständen übersetzt :-)
- Der Code hinter der Schleife muss gegenüber dem Endzustand jedes Sprungs aus der Schleife korrekt sein!

Beispiel:

$$\text{for} \quad (x = 0; x < n; x{+}{+})$$
$$M[A + x] = z;$$



Verdoppelung des Rumpfs liefert:

$$\text{for } (x = 0; x < n; x{+}{+}) \ \{$$

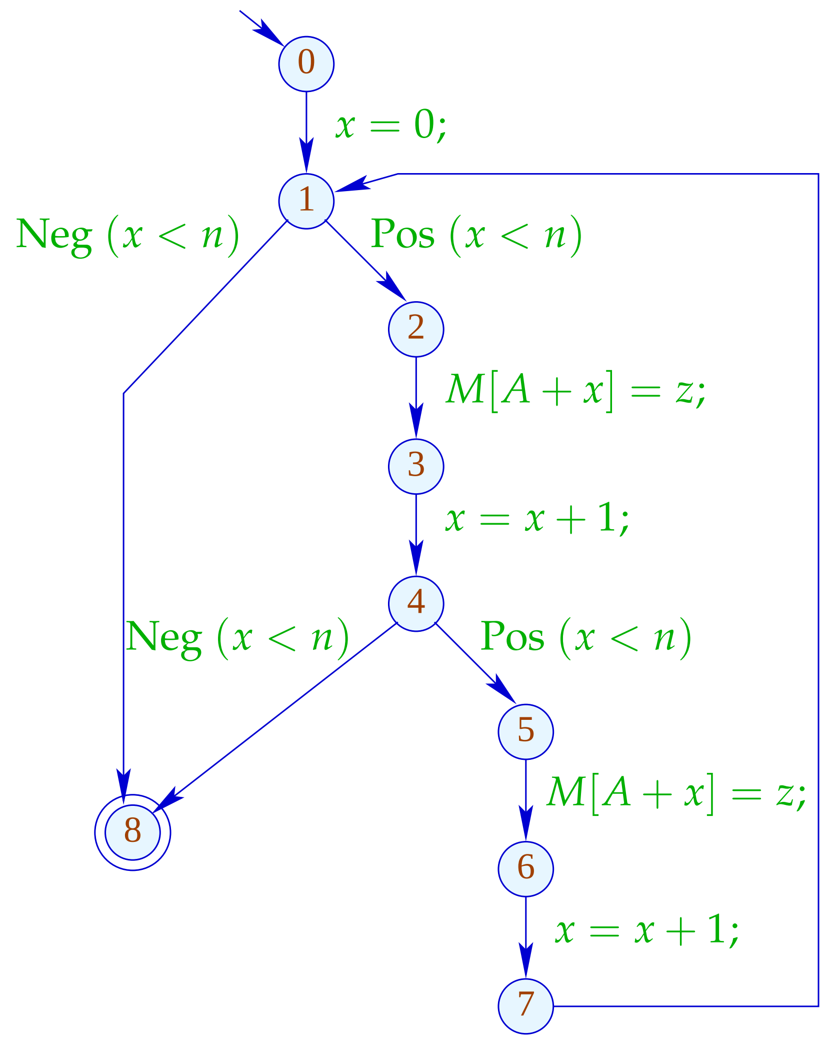
$$M[A + x] = z;$$

$$x = x + 1;$$

$$\text{if } (!(x < n)) \ \text{break};$$

$$M[A + x] = z;$$

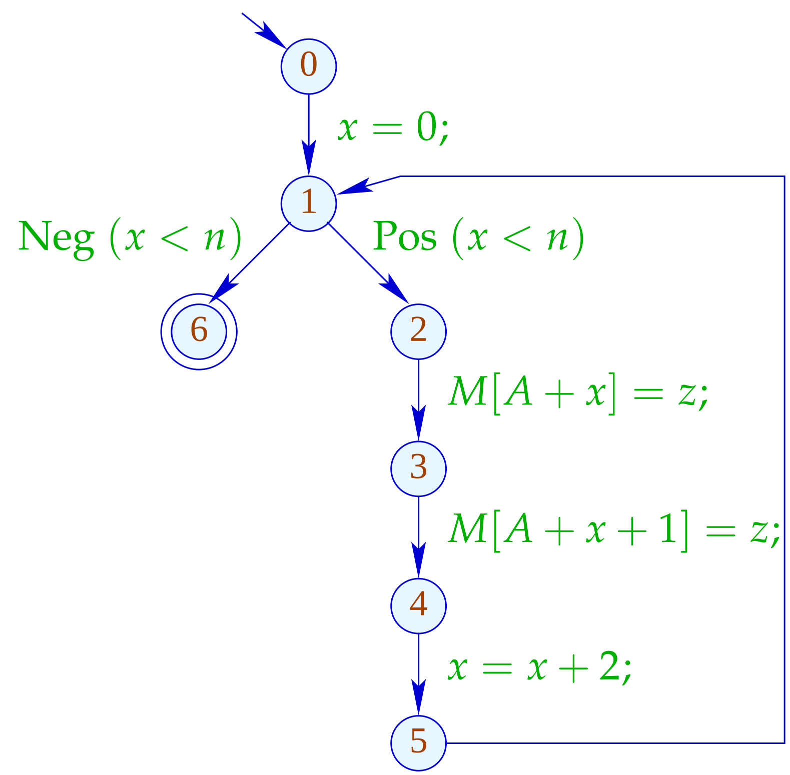
$$\}$$



Besser wäre es, wenn wir auf den Test in der Mitte verzichten könnten. Das ist möglich, wenn wir wissen, dass n stets gerade ist :-)

Dann haben wir:

$$\text{for } (x = 0; x < n; x = x + 2) \; \{$$
$$M[A + x] = z;$$
$$M[A + x + 1] = z;$$
$$\}$$



Diskussion:

- Beseitigung der Zwischenabfrage zusammen mit Verschieben des Zwischen-Inkrements ans Ende zeigt, dass die verschiedenen Rumpf-Iterationen in Wahrheit unabhängig sind   :-)
- Wir gewinnen trotzdem nicht viel, da wir nur maximal ein Store pro Wort gestatten   :-(
- Sind die rechten Seiten allerdings komplizierter, könnten wir deren Auswertung mit je einem Store pro Takt verschränken :-)

Erweiterung 3:

Möglicherweise bietet eine Schleife allein nicht genug Möglichkeiten zur Parallelisierung :-(

... möglicherweise aber zwei aufeinander folgende :-)

Beispiel:

$$\begin{array}{ll}
\text{for } (x = 0; x < n; x{+}{+})\ \{ & \text{for } (x = 0; x < n; x{+}{+})\ \{ \\
\qquad R = M[B + x]; & \qquad R = M[B + x]; \\
\qquad S = M[C + x]; & \qquad S = M[C + x]; \\
\qquad T_1 = R + S; & \qquad T_2 = R - S; \\
\qquad M[A + x] = T_1; & \qquad M[C + x] = T_2; \\
\} & \}
\end{array}$$

Um beide Schleifen zu einer zusammen zu fassen, muss:

- das Iterations-Schema übereinstimmen;
- die beiden Schleifen greifen auf unterschiedliche Daten zu.

Im Falle von einzelnen Variablen lässt sich das leicht verfizieren.

Schwieriger ist das in Anwesenheit von Pointern oder Feldern.

Unter Rückgriff auf das Source-Programm kann man Zugriffe auf statisch allokierte disjunkte Felder erkennen.

Analyse von Zugriffen auf das gleiche Feld ist erheblich schwieriger ...

Nehmen wir für das Beispiel an, die Bereiche
$[A, A+n-1], [B, B+n-1], [C, C+n-1]$ überlappen nicht.

Offenbar können wir dann die beiden Schleifen kombinieren zu:

$$
\begin{array}{l}
\textsf{for}\ \ (x = 0; x < n; x{+}{+})\ \ \{ \\
\qquad \begin{array}{ll}
R = M[B+x]; & R = M[B+x]; \\
S = M[C+x]; & S = M[C+x]; \\
T_1 = R + S; & T_2 = R - S; \\
M[A+x] = T_1; & M[C+x] = T_2;
\end{array} \\
\}
\end{array}
$$

Die erste Schleife darf in Iteration x auf keine Daten zugreifen, die die zweite Schleife in Iterationen $< x$ modifiziert.

Die zweite Schleife darf in Iteration x auf keine Daten zugreifen, die die erste Schleife in Iterationen $> x$ überschreibt.

I.a. muss man dazu die Indexausdrücke analysieren.

Sind diese linear, führt das auf Probleme des integer linear programming:

$$
\begin{aligned}
x_{\text{write}} &\geq C \\
x_{\text{write}} &\leq C + x - 1 \\
x_{\text{read}} &= C + x \\
x_{\text{read}} &= x_{\text{write}}
\end{aligned}
$$

... hat offenbar keine Lösung :-)

Allgemeine Form:

$$\begin{array}{rcl} x & \geq & t_1 \\ t_2 & \geq & x \\ y & = & s \\ x & = & y \end{array}$$

für lineare Ausdrücke s, t_1, t_2 über den Iterations-Variablen.

Das lässt sich vereinfachen zu:

$$0 \leq s - t_1 \qquad\qquad 0 \leq t_2 - s$$

Was macht man damit ???

Einfacher Fall:

Die beiden Ungleichungen haben über $\mathbb{Q}$ eine leere Lösungsmenge.

Dann ist die Lösungsmenge auch über $\mathbb{Z}$ leer :-)

In unserem Beispiel:

$$\begin{array}{lll} 0 & \leq & C + x - C & = & x \\ 0 & \leq & C + x - 1 - (C + x) & = & -1 \end{array}$$

Die zweite Ungleichung hat überhaupt keine Lösung :-)

Gleiche Vorzeichen:

Kommt eine Variable x in allen Ungleichungen mit gleichem Vorzeichen vor, gibt es immer eine Lösung :-(

Beispiel:

$$
\begin{array}{rcl}
0 & \leq & 13 + 7 \cdot x \\
0 & \leq & -1 + 5 \cdot x
\end{array}
$$

Man muss x nur wählen als:

$$x \geq \mathsf{max}(-\frac{13}{7}, \frac{1}{5}) = \frac{1}{5}$$

Ungleiche Vorzeichen:

Eine Variable x kommt in einer Ungleichung negativ, in allen anderen höchstens positiv vor. Dann kann man ein Ungleichungssystem ohne x konstruieren ...

Beispiel:

$$\begin{array}{lll} 0 & \leq & 13 - 7 \cdot x \\ 0 & \leq & -1 + 5 \cdot x \end{array} \quad \Longleftrightarrow \quad \begin{array}{lll} x & \leq & \frac{13}{7} \\ 0 & \leq & -1 + 5 \cdot x \end{array}$$

Da $0 \leq -1 + 5 \cdot \frac{13}{7}$ hat das System eine rationale Lösung ...

Eine Variable:

Die Ungleichungen, in denen x positiv vorkommt, liefern untere Schranken.

Die Ungleichungen, in denen x negativ vorkommt, liefern obere Schranken.

Seien G, L die grösste untere bzw. kleinste obere Schranke. Dann liegen alle (ganzzahligen) Lösungen im Intervall $[G, L]$:-)

Beispiel:

$$\begin{array}{rcl} 0 & \leq & 13 - 7 \cdot x \\ 0 & \leq & -1 + 5 \cdot x \end{array} \quad \Longleftrightarrow \quad \begin{array}{rcl} x & \leq & \frac{13}{7} \\ x & \geq & \frac{1}{5} \end{array}$$

Die einzige ganzzahlige Lösung des Systems ist $x = 1$:-)

Diskussion:

- Lösungen sind natürlich immer nur innerhalb der Grenzen der Iterationsvariablen interessant.
- Jede ganzzahlige Lösung dort liefert einen Konflikt.
- Verschränkte Berechnung der Schleifen ist möglich, sofern es keinerlei Konflikte gibt :-)
- Die angegebenen Spezialfälle reichen, um den Fall von zwei Ungleichungen über $\mathbb{Q}$ bzw. einer Variable über $\mathbb{Z}$ zu behandeln.
- Die Anzahl der Variablen in den Ungleichungen entspricht der Anzahl der geschachtelten for-Schleifen $\Longrightarrow$ sie ist i.a. klein :-)

Diskussion:

- Integer Linear Programming (ILP) kann die Erfüllbarkeit herausfinden einer endlichen Menge von Gleichungen/Ungleichungen über $\mathbb{Z}$ der Form:

$$\sum_{i=1}^{n} a_i \cdot x_i = b \quad \text{bzw.} \quad \sum_{i=1}^{n} a_i \cdot x_i \geq b\,, \quad a_i \in \mathbb{Z}$$

- Darüber hinaus kann eine (lineare) Zielfunktion optimiert werden :-)
- Achtung: Bereits das Entscheidungsproblem ist i.a. NP-schwierig !!!
- Trotzdem gibt es erstaunlich effiziente Implementierungen.
- Nicht nur Schleifen-Verschmelzung, auch andere Umstrukturierungen von Schleifen führen auf ILP-Probleme ...

Exkurs 5: Presburger Arithmetik

Viele Probleme der Informatik lassen sich ohne Multiplikation formulieren :-)

Wir betrachten hier erst einmal zwei einfache Spezialfälle ...

1. Lineare Gleichungen

$$
\begin{array}{rcrcrcr}
2x & + & 3y & & & = & 24 \\
x & - & y & + & 5z & = & 3
\end{array}
$$

Fragen:

- Gibt es eine Lösung über $\mathbb{Q}$?
- Gibt es eine Lösung über $\mathbb{Z}$?
- Gibt es eine Lösung über $\mathbb{N}$?

Schauen wir uns dazu nochmal die Gleichungen an:

$$\begin{array}{rcrcrcr} 2x & + & 3y & & & = & 24 \\ x & - & y & + & 5z & = & 3 \end{array}$$

Antworten:

- Gibt es eine Lösung über $\mathbb{Q}$? Ja
- Gibt es eine Lösung über $\mathbb{Z}$? Nein
- Gibt es eine Lösung über $\mathbb{N}$? Nein

Komplexität:

- Gibt es eine Lösung über $\mathbb{Q}$? polynomiell
- Gibt es eine Lösung über $\mathbb{Z}$? polynomiell
- Gibt es eine Lösung über $\mathbb{N}$? NP-schwierig

Lösungsverfahren für Integers

Beobachtung 1:

$$a_1 x_1 + \ldots + a_k x_k = b \qquad\qquad (\forall\, i : \; a_i \neq 0)$$

hat eine Lösung genau dann wenn

$$\text{ggT}\{a_1, \ldots, a_k\} \quad \mid \quad b$$

Beispiel:

$$5y - 10z = 18$$

hat keine Lösung über $\mathbb{Z}$:-)

Beispiel:

$$5y - 10z = 18$$

hat keine Lösung über $\mathbb{Z}$:-)

Beobachtung 2:

Eine Variable mit Koeefizient ± 1 kann beseitigt werden.

Beispiel:

$$\begin{array}{rcrcrcr} 2x & + & 3y & & & = & 24 \\ x & - & y & + & 5z & = & 3 \end{array}$$

Beispiel:

$$\begin{array}{rcrcrcr} 2x & + & 3y & & & = & 24 \\ x & - & y & + & 5z & = & 3 \end{array}$$

Beispiel:

$$\begin{array}{rcrcrcr} 2x & + & 3y & & & = & 24 \\ x & - & y & + & 5z & = & 3 \end{array}$$

$$\Longrightarrow \qquad x = 3 + y - 5z$$

Beispiel:

$$
\begin{aligned}
2x + 3y \qquad\qquad &= 24 \\
x &= 3 + y - 5z
\end{aligned}
$$

Beispiel:

$$2x + 3y = 24$$

$$\Longrightarrow \quad x = 3 + y - 5z$$

$$5y - 10z = 18$$

Beobachtung 3:

Jede (lösbare) Gleichung kann so massiert werden, dass sie eine Variable mit Koeffizient ± 1 besitzt :-)

Beobachtung 3:

Jede (lösbare) Gleichung kann so massiert werden, dass sie eine Variable mit Koeffizient ± 1 besitzt :-)

... mithilfe von uni-modularen Variablentransformationen :-))

Nehmen wir an, die Gleichung enthalte $a_1 x_1 + a_2 x_2$ mit

$$\text{ggT}\{a_1, a_2\} = p$$

Idee:

Ersetze x_1, x_2 durch zwei neue Variablen t_1, t_2 so dass zum Einen gilt:

$$\begin{array}{rcccc} p t_1 & = & a_1 x_1 & + & a_2 x_2 \\ t_2 & = & b_1 x_1 & + & b_2 x_2 \end{array}$$

für geeignete b_1, b_2 ...

Nehmen wir an, die Gleichung enthalte $a_1 x_1 + a_2 x_2$ mit

$$\text{ggT}\{a_1, a_2\} = p$$

Idee:

Ersetze x_1, x_2 durch zwei neue Variablen t_1, t_2 so dass zum Einen gilt:

$$\begin{array}{rcrcr} pt_1 & = & a_1 x_1 & + & a_2 x_2 \\ t_2 & = & b_1 x_2 & + & b_2 x_2 \end{array}$$

für geeignete b_1, b_2 ... und zum Anderen,

alle Lösungen für t_1, t_2 auch Lösungen für x_1, x_2 ergeben

:-)

$\Longrightarrow$ Die inverse Matrix der Transformation:

$$\begin{pmatrix} \frac{a_1}{p} & \frac{a_2}{p} \\ b_1 & b_2 \end{pmatrix}$$

sollte ganzzahlige Koeffizienten haben.

$\Longrightarrow$ Die inverse Matrix der Transformation:

$$\begin{pmatrix} \frac{a_1}{p} & \frac{a_2}{p} \\ b_1 & b_2 \end{pmatrix}$$

sollte ganzzahlige Koeffizienten haben.

Dies ist der Fall, wenn

$$\frac{a_1}{p} b_2 - \frac{a_2}{p} b_1 \quad = \quad \pm 1$$

Da a_1, a_2 den ggT p haben,
findet Euclid's Algo λ_1, λ_2 mit:

$$a_1\lambda_1 + a_2\lambda_2 = p$$

Da a_1, a_2 den ggT p haben,
findet Euclid's Algo λ_1, λ_2 mit:

$$a_1\lambda_1 + a_2\lambda_2 \quad = \quad p$$

$\Longrightarrow$

Wähle: $b_1 = -\lambda_2 \qquad b_2 = \lambda_1.$

Da a_1, a_2 den ggT p haben,
findet Euclid's Algo λ_1, λ_2 mit:

$$a_1\lambda_1 + a_2\lambda_2 = p$$

$\Longrightarrow$

Wähle: $b_1 = -\lambda_2 \qquad b_2 = \lambda_1.$

Dann:

$$\begin{aligned} x_1 &= \lambda_1 t_1 - \tfrac{a_2}{p} t_2 \\ x_2 &= \lambda_2 t_1 + \tfrac{a_1}{p} t_2 \end{aligned}$$

Beispiel:

$$
\begin{array}{rcrcrcr}
-2x_1 & + & 5x_2 & + & 3x_3 & = & 2 \\
-4x_1 & + & 3x_2 & - & 2x_3 & = & -1
\end{array}
$$

Beispiel:

$$
\begin{array}{rcrcrcr}
-2x_1 & + & 5x_2 & + & 3x_3 & = & 2 \\
-4x_1 & + & 3x_2 & - & 2x_3 & = & -1
\end{array}
$$

Beispiel:

$$\begin{array}{rcrcrcr} -2x_1 & + & 5x_2 & + & 3x_3 & = & 2 \\ -4x_1 & + & 3x_2 & - & 2x_3 & = & -1 \end{array}$$

Euclid: $\lambda_1 = -1 \quad \lambda_2 = -1$

Beispiel:

$$\begin{array}{rcrcrcr} -2x_1 & + & 5x_2 & + & 3x_3 & = & 2 \\ -4x_1 & + & 3x_2 & - & 2x_3 & = & -1 \end{array}$$

Euclid: $\lambda_1 = -1 \quad \lambda_2 = -1$

$\Longrightarrow$

$$\begin{array}{rcrcr} x_1 & = & -t_1 & - & 3t_2 \\ x_2 & = & -t_1 & - & 4t_2 \end{array}$$

Ersetzen vom x_1, x_2 mit t_1, t_2 liefert:

$$\begin{array}{rcrcrcr} -7t_1 & - & 26t_2 & + & 3x_3 & = & 2 \\ t_1 & & & - & 2x_3 & = & -1 \end{array}$$

... und wir haben eine Variable beseitigt :-)

Lösen über $\mathbb{N}$

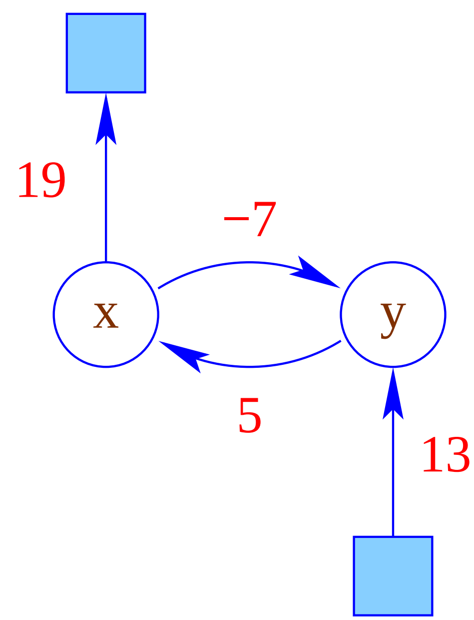
- ... ist von großer praktischer Bedeutung;
- ... hat zur Entwicklung vieler neuer Techniken geführt;
- ... erlaubt leicht die Kodierung NP-schwieriger Probleme;
- ... bleibt schwierig, sogar wenn nur drei Variablen pro Gleichung erlaubt sind.

2. Ein polynomieller Spezialfall:

$$\begin{array}{rcl} & x & \geq & y+5 \\ 19 & \geq & x & \\ & y & \geq & 13 \\ & y & \geq & x-7 \end{array}$$

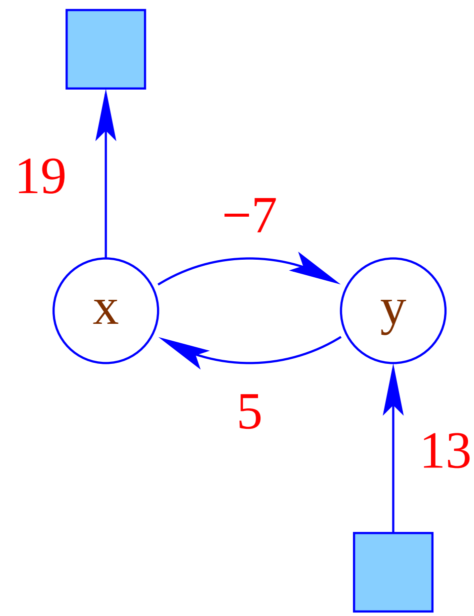
- Es gibt maximal zwei Variablen pro Un-Gleichung;
- keine Skalierungsfaktoren.

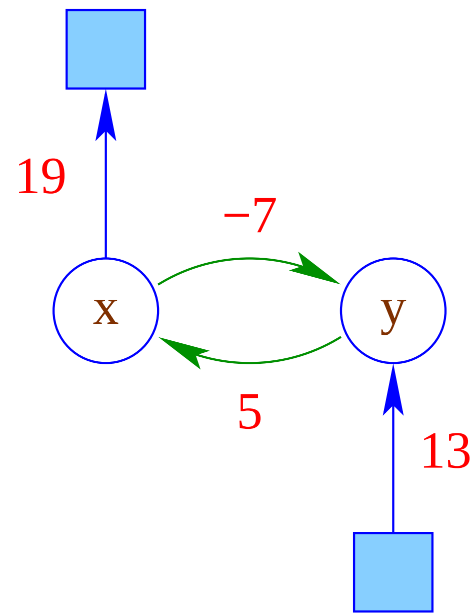
Idee: Representiere das System als Graph:

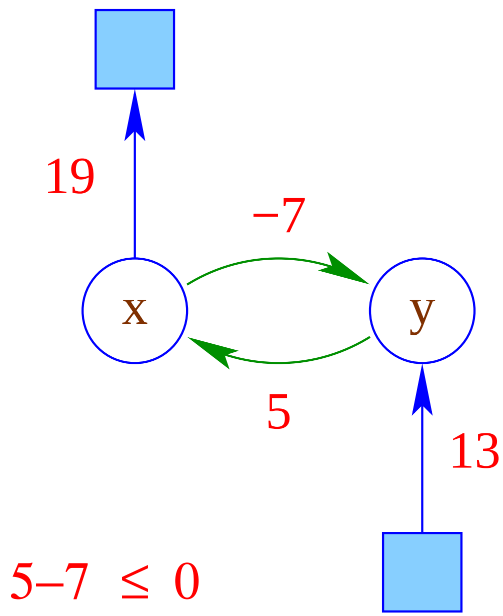


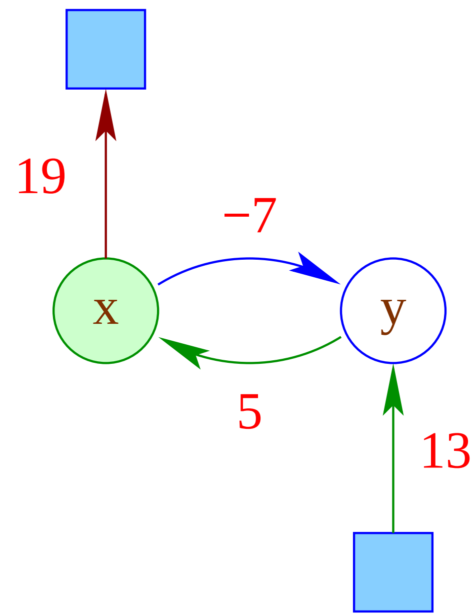
Die Ungleichungen sind erfüllbar genau dann wenn

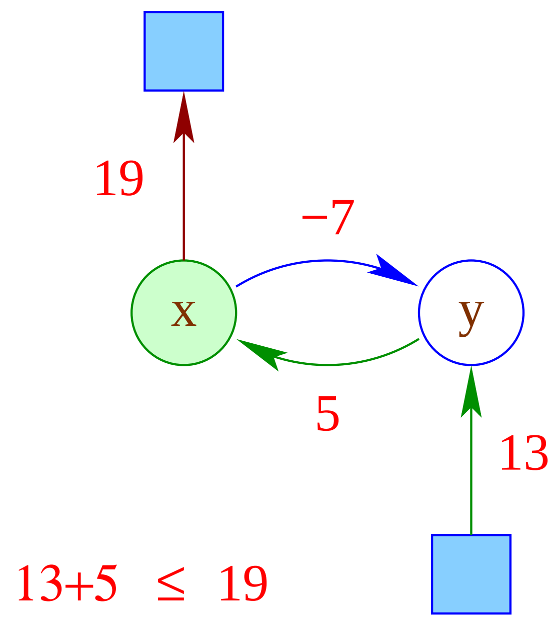
- die Gewichte jedes Kreises maximal ≤ 0 sind;
- die Gewichte, die x erreichen, maximal $\leq$ der Gewichte sind, die x verlassen.


19
−7
x
y
5
13


19
−7
x
y
5
13


19
−7
x
y
5
13
5−7 ≤ 0


19
−7
x
y
5
13


19
−7
x
y
5
13
13+5 ≤ 19

Die Ungleichungen sind erfüllbar genau dann wenn

- die Gewichte jedes Kreises maximal ≤ 0 sind;
- die Gewichte, die x erreichen, maximal $\leq$ der Gewichte sind, die x verlassen.

reflexive und transitive Hülle der Kanten-Gewichte!

3. Ein allgemeines Lösungsverfahren:

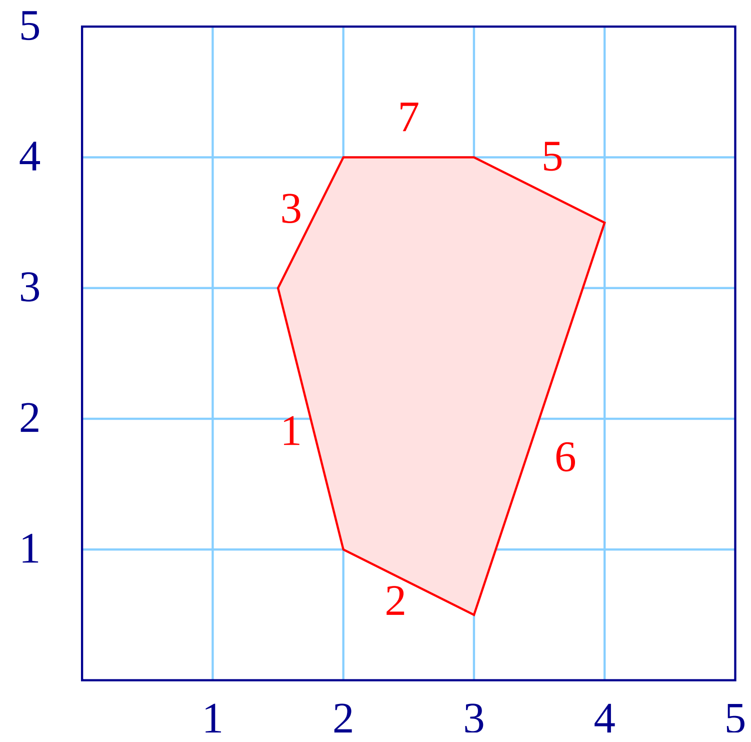
Idee: Fourier-Motzkin-Elimination

- Beseitige sukzessive einzelne Variablen x !
- Alle Ungleichungen mit positiven Vorkommen von x liefern untere Schranken.
- Alle Ungleichungen mit negativen Vorkommen von x liefern obere Schranken.
- Alle unteren Schranken müssen kleiner oder gleich allen oberen Schranken sein ;-))

Jean Baptiste Joseph Fourier, 1768–1830

Beispiel:

$$
\begin{array}{rcll}
9 & \leq & 4x_1 + x_2 & (1) \\
4 & \leq & x_1 + 2x_2 & (2) \\
0 & \leq & 2x_1 - x_2 & (3) \\
6 & \leq & x_1 + 6x_2 & (4) \\
-11 & \leq & -x_1 - 2x_2 & (5) \\
-17 & \leq & -6x_1 + 2x_2 & (6) \\
-4 & \leq & -x_2 & (7)
\end{array}
$$



Für x_1 finden wir:

$$\begin{array}{rcll} 9 & \leq & 4x_1 + x_2 & (1) \\ 4 & \leq & x_1 + 2x_2 & (2) \\ 0 & \leq & 2x_1 - x_2 & (3) \\ 6 & \leq & x_1 + 6x_2 & (4) \\ -11 & \leq & -x_1 - 2x_2 & (5) \\ -17 & \leq & -6x_1 + 2x_2 & (6) \\ -4 & \leq & -x_2 & (7) \end{array} \qquad \begin{array}{rcll} \frac{9}{4} - \frac{1}{4}x_2 & \leq & x_1 & (1) \\ 4 - 2x_2 & \leq & x_1 & (2) \\ \frac{1}{2}x_2 & \leq & x_1 & (3) \\ 6 - 6x_2 & \leq & x_1 & (4) \\ x_1 & \leq & 11 - 2x_2 & (5) \\ x_1 & \leq & \frac{17}{6} + \frac{1}{3}x_2 & (6) \\ -4 & \leq & -x_2 & (7) \end{array}$$

Wenn es ein solches x_1 gibt, müssen alle unteren Schranken kleiner gleich allen oberen sein, d.h.:

$$
\begin{array}{rcll}
\frac{9}{4} - \frac{1}{4}x_2 & \leq & 11 - 2x_2 & (1,5) \\
\frac{9}{4} - \frac{1}{4}x_2 & \leq & \frac{17}{6} + \frac{1}{3}x_2 & (1,6) \\
4 - 2x_2 & \leq & 11 - 2x_2 & (2,5) \\
4 - 2x_2 & \leq & \frac{17}{6} + \frac{1}{3}x_2 & (2,6) \\
\frac{1}{2}x_2 & \leq & 11 - 2x_2 & (3,5) \\
\frac{1}{2}x_2 & \leq & \frac{17}{6} + \frac{1}{3}x_2 & (3,6) \\
6 - 6x_2 & \leq & 11 - 2x_2 & (4,5) \\
6 - 6x_2 & \leq & \frac{17}{6} + \frac{1}{3}x_2 & (4,6) \\
-4 & \leq & -x_2 & (7)
\end{array}
$$

oder:

$$
\begin{array}{rcll}
-35 & \leq & -7x_2 & (1,5) \\
-\frac{7}{12} & \leq & \frac{7}{12}x_2 & (1,6) \\
-7 & \leq & 0 & (2,5) \\
\frac{7}{6} & \leq & \frac{7}{3}x_2 & (2,6) \\
-22 & \leq & -5x_2 & (3,5) \\
-\frac{17}{6} & \leq & -\frac{1}{6}x_2 & (3,6) \\
-5 & \leq & 4x_2 & (4,5) \\
\frac{19}{6} & \leq & \frac{19}{3}x_2 & (4,6) \\
-4 & \leq & -x_2 & (7)
\end{array}
$$

$$
\begin{array}{rcll}
\frac{9}{4} - \frac{1}{4}x_2 & \leq & 11 - 2x_2 & (1,5) \\
\frac{9}{4} - \frac{1}{4}x_2 & \leq & \frac{17}{6} + \frac{1}{3}x_2 & (1,6) \\
4 - 2x_2 & \leq & 11 - 2x_2 & (2,5) \\
4 - 2x_2 & \leq & \frac{17}{6} + \frac{1}{3}x_2 & (2,6) \\
\frac{1}{2}x_2 & \leq & 11 - 2x_2 & (3,5) \\
\frac{1}{2}x_2 & \leq & \frac{17}{6} + \frac{1}{3}x_2 & (3,6) \\
6 - 6x_2 & \leq & 11 - 2x_2 & (4,5) \\
6 - 6x_2 & \leq & \frac{17}{6} + \frac{1}{3}x_2 & (4,6) \\
-4 & \leq & -x_2 & (7)
\end{array}
\qquad \text{oder:} \qquad
\begin{array}{rcll}
-5 & \leq & -x_2 & (1,5) \\
-1 & \leq & x_2 & (1,6) \\
-7 & \leq & 0 & (2,5) \\
\frac{1}{2} & \leq & x_2 & (2,6) \\
-\frac{22}{5} & \leq & -x_2 & (3,5) \\
-17 & \leq & -x_2 & (3,6) \\
-\frac{5}{4} & \leq & x_2 & (4,5) \\
\frac{1}{2} & \leq & x_2 & (4,6) \\
-4 & \leq & -x_2 & (7)
\end{array}
$$

Das ist der Ein-Variablen-Fall, den wir exakt lösen können:

$$\max\{-1, \boxed{\tfrac{1}{2}}, -\tfrac{5}{4}, \tfrac{1}{2}\} \quad \leq \quad x_2 \quad \leq \quad \min\{5, \tfrac{22}{5}, 17, \boxed{4}\}$$

Daraus können wir folgern: $\qquad x_2 \in [\frac{1}{2}, 4] \qquad$:-)

Im Allgemeinen:

- Das ursprüngliche System hat eine Lösung in $\mathbb{Q}$, gdw. das System nach Eliminierung einer Variable eine Lösung in $\mathbb{Q}$ besitzt :-)
- In jedem Eliminierungsschritt kann sich die Anzahl der Ungleichungen quadrieren $\Longrightarrow$ exponentielle Laufzeit :-((
- Es lässt sich so modifizieren, dass es Erfüllbarkeit über $\mathbb{Z}$ entscheidet $\Longrightarrow$ Omega-Test

William Worthington Pugh, Jr.
University of Maryland, College Park

Idee:

- Wir beseitigen sukzessive die Variablen. Dabei müssen wir allerdings Divisionen vermeiden ...
- Hat x überall Koeffizienten ± 1, machen wir Fourier-Motzkin-Elimination :-)
- Andernfalls stellen wir x auf einer Seite mit positivem Koeffizienten frei ...

Betrachten wir etwa (1) und (6) :

$$\begin{array}{rcl} 6 \cdot x_1 & \leq & 17 + 2x_2 \\ 9 - x_2 & \leq & 4 \cdot x_1 \end{array}$$

E.O. können wir echte Ungleichungen betrachten:

$$\begin{aligned} 6 \cdot x_1 &< 18 + 2x_2 \\ 8 - x_2 &< 4 \cdot x_1 \end{aligned}$$

... und jeweils durch den ggT teilen:

$$\begin{aligned} 3 \cdot x_1 &< 9 + x_2 \\ 8 - x_2 &< 4 \cdot x_1 \end{aligned}$$

Das impliziert:

$$3 \cdot (8 - x_2) < 4 \cdot (9 + x_2)$$

Offenbar gilt:

- Ist die abgeleitete Ungleichung unerfüllbar, dann das ganze System :-)
- Sind alle so abgeleiteten Ungleichungen erfüllbar, gibt es eine Lösung, die aber möglicherweise nicht ganzzahlig ist :-(
- Es gibt aber eine ganzzahlige Lösung, sofern zwischen unterer und oberer Schranke stets genug Platz ist, so dass ein Integer dazwischen passt.
- Sei $\alpha < a \cdot x \qquad b \cdot x < \beta$.

  Dann muss nicht nur gelten:

  $$b \cdot \alpha < a \cdot \beta$$

  sondern sogar

  $$\boxed{a \cdot b} < a \cdot \beta - b \cdot \alpha$$

... im Beispiel:

$$12 \quad < \quad 4 \cdot (9 + x_2) - 3 \cdot (8 - x_2)$$

oder:

$$12 \quad < \quad 12 + 7x_2$$

bzw:

$$0 \quad < \quad x_2$$

Im Beispiel lassen sich auch diese verschärften Ungleichungen erfüllen

$\Longrightarrow$ das System hat über $\mathbb{Z}$ eine Lösung :-)

Überlegung:

- Sind die verschärften Ungleichungen erfüllbar, dann auch das ursprüngliche System. Die Umkehrung gilt i.A. nicht :-(
- In dem Fall ist bei einem Paar Ungleichungen weniger Platz:

$$a \cdot \beta \leq b \cdot \alpha + \boxed{a \cdot b}$$

oder:

$$b \cdot \alpha < ab \cdot x < b \cdot \alpha + \boxed{a \cdot b}$$

Kürzen durch b liefert:

$$\alpha < a \cdot x < \alpha + \boxed{a}$$

$\Longrightarrow$ $\boxed{\alpha + i = a \cdot x}$ für ein $i \in \{1, \ldots, a - 1\}$!!!

Diskussion:

- $\longrightarrow$ Fourier-Motzkin-Elimination ist nicht das beste Verfahren für rationale Ungleichungssysteme.
- $\longrightarrow$ Der Omega-Test ist notwendig exponentiell :-)

  Wenn das System lösbar ist, terminiert der Test i.a. schnell.

  Mit unlösbaren Systemen tut er sich schwerer :-(
- $\longrightarrow$ Auch für ILP gibt es andere/intelligentere Verfahren ...
- $\longrightarrow$ Für Probleme bei Programmiersprachen funktioniert er wohl ganz gut :-)

4. Verallgemeinerung zu einer Logik

Disjunktion:

$$
\begin{array}{llll}
(x - 2y = 15 & \wedge & x + y = 7) & \vee \\
(x + y = 6 & \wedge & 3x + z = -8) &
\end{array}
$$

Quantoren:

$$
\exists\, x : \quad z - 2x = 42 \quad \wedge \quad z + x = 19
$$

4. Verallgemeinerung zu einer Logik

Disjunktion:

$$(x - 2y = 15 \quad \wedge \quad x + y = 7) \qquad \vee$$
$$(x + y = 6 \qquad \wedge \quad 3x + z = -8)$$

Quantoren:

$$\exists\, x : \quad z - 2x = 42 \quad \wedge \quad z + x = 19$$

$\Longrightarrow$ Presburger Arithmetik

Mojzesz Presburger, 1904–1943 (?)

Presburger Arithmetik $=$ normale Arithmetik ohne Multiplikation

Presburger Arithmetik	==	normale Arithmetik ohne Multiplikation
Arithmetik	:	hochgradig unentscheidbar :-(sogar sogar unvollständig :-((

Presburger Arithmetik $=$ normale Arithmetik
ohne Multiplikation

Arithmetik : hochgradig unentscheidbar :-(
sogar sogar unvollständig :-((

$\Longrightarrow$ Hilberts 10tes Problem

$\Longrightarrow$ Gödels Theorem

Presburger Formeln:

$$
\begin{array}{lcl}
\phi & ::= & x + y = z \quad | \quad x = n \quad | \\
 & & \phi_1 \wedge \phi_2 \quad | \quad \neg \phi \quad | \\
 & & \exists\, x : \ \ \phi
\end{array}
$$

Presburger Formeln:

$$\begin{array}{lcl} \phi & ::= & x + y = z \quad | \quad x = n \quad | \\ & & \phi_1 \wedge \phi_2 \quad | \quad \neg\phi \quad | \\ & & \exists\, x : \ \phi \end{array}$$

Ziel: PSAT

Finde Werte in $\mathbb{N}$ für die freien Variablen, so dass ϕ gilt ...

Idee: Codiere die Werte der Variablen als Worte :-)

213	t	1	0	1	0	1	0	1	1
42	z	0	1	0	1	0	1	0	0
89	y	1	0	0	1	1	0	1	0
17	x	1	0	0	0	1	0	0	0

Idee: Codiere die Werte der Variablen als Worte :-)

213	t	1	0	1	0	1	0	1	1
42	z	0	1	0	1	0	1	0	0
89	y	1	0	0	1	1	0	1	0
17	x	1	0	0	0	1	0	0	0

Idee: Codiere die Werte der Variablen als Worte :-)

213	t	1	0	1	0	1	0	1	1
42	z	0	1	0	1	0	1	0	0
89	y	1	0	0	1	1	0	1	0
17	x	1	0	0	0	1	0	0	0

Idee: Codiere die Werte der Variablen als Worte :-)

213	t	1	0	1	0	1	0	1	1
42	z	0	1	0	1	0	1	0	0
89	y	1	0	0	1	1	0	1	0
17	x	1	0	0	0	1	0	0	0

Idee: Codiere die Werte der Variablen als Worte :-)

213	t	1	0	1	0	1	0	1	1
42	z	0	1	0	1	0	1	0	0
89	y	1	0	0	1	1	0	1	0
17	x	1	0	0	0	1	0	0	0

Idee: Codiere die Werte der Variablen als Worte :-)

213	t	1	0	1	0	1	0	1	1
42	z	0	1	0	1	0	1	0	0
89	y	1	0	0	1	1	0	1	0
17	x	1	0	0	0	1	0	0	0

Idee: Codiere die Werte der Variablen als Worte :-)

213	t	1	0	1	0	1	0	1	1
42	z	0	1	0	1	0	1	0	0
89	y	1	0	0	1	1	0	1	0
17	x	1	0	0	0	1	0	0	0

Idee: Codiere die Werte der Variablen als Worte :-)

213	t	1	0	1	0	1	0	1	1
42	z	0	1	0	1	0	1	0	0
89	y	1	0	0	1	1	0	1	0
17	x	1	0	0	0	1	0	0	0

Idee: Codiere die Werte der Variablen als Worte :-)

213	t	1	0	1	0	1	0	1	1
42	z	0	1	0	1	0	1	0	0
89	y	1	0	0	1	1	0	1	0
17	x	1	0	0	0	1	0	0	0

Idee: Codiere die Werte der Variablen als Worte :-)

213	t	1	0	1	0	1	0	1	1
42	z	0	1	0	1	0	1	0	0
89	y	1	0	0	1	1	0	1	0
17	x	1	0	0	0	1	0	0	0

Beobachtung:

Die Menge der erfüllenden Variablenbelegungen ist regulär :-))

Beobachtung:

Die Menge der erfüllenden Variablenbelegungen ist regulär :-))

$$
\begin{array}{lcll}
\phi_1 \wedge \phi_2 & \Longrightarrow & \mathcal{L}(\phi_1) \cap \mathcal{L}(\phi_2) & \text{(Durchschnitt)} \\
\neg\phi & \Longrightarrow & \overline{\mathcal{L}(\phi)} & \text{(Komplement)} \\
\exists\, x : \quad \phi & \Longrightarrow & \pi_x(\mathcal{L}(\phi)) & \text{(Projektion)}
\end{array}
$$

Weg-Projizierung der x-Komponente:

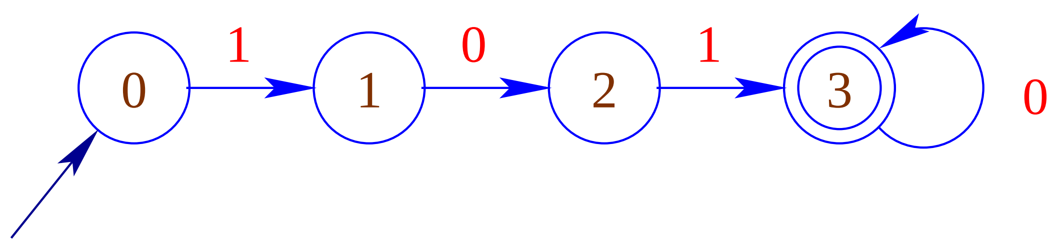
213	t	1	0	1	0	1	0	1	1
42	z	0	1	0	1	0	1	0	0
89	y	1	0	0	1	1	0	1	0
17	x	1	0	0	0	1	0	0	0

Weg-Projizierung der x-Komponente:

213	t	1	0	1	0	1	0	1	1
42	z	0	1	0	1	0	1	0	0
89	y	1	0	0	1	1	0	1	0

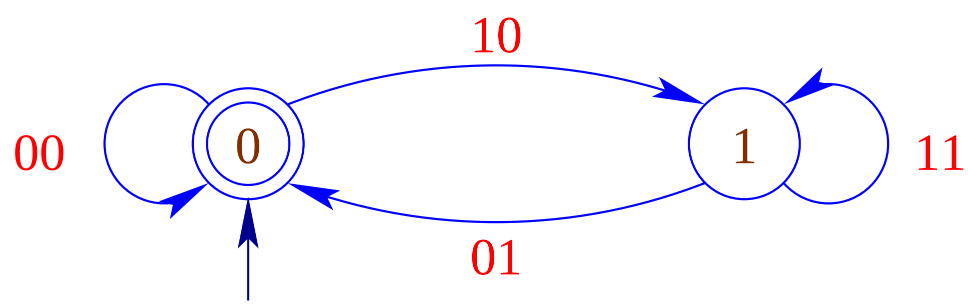
Automaten für Basis-Prädikate:

$x = 5$



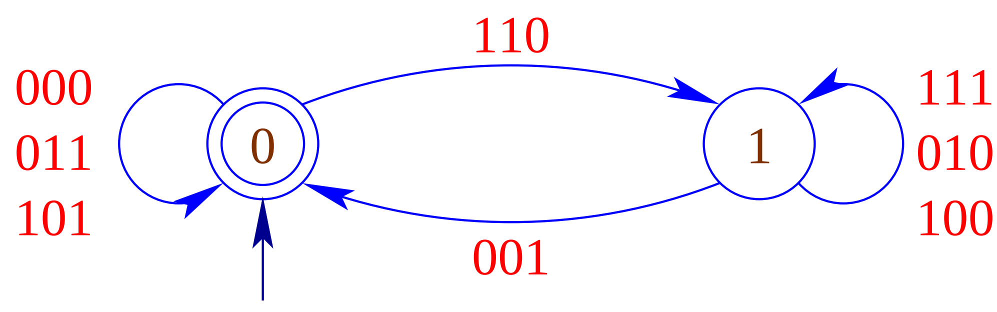
Automaten für Basis-Prädikate:

$$x+x = y$$



Automaten für Basis-Prädikate:

$x+y = z$



Ergebnisse:

Ferrante, Rackoff,1973 : $\text{PSAT} \leq \text{DSPACE}(2^{2^{c \cdot n}})$

Ergebnisse:

Ferrante, Rackoff,1973 : $\qquad \text{PSAT} \leq \text{DSPACE}(2^{2^{c \cdot n}})$

Fischer, Rabin,1974 : $\qquad \text{PSAT} \geq \text{NTIME}(2^{2^{c \cdot n}})$

3.4 Verbesserung der Speicher-Organisation

Ziel:

- Ausnutzung von Caches

  $\Longrightarrow$ Verringerung der Anzahl der Cache-Misses
- Verringerung der Allokations / Deallokations-Kosten

  $\Longrightarrow$ Ersetzung von Heap-Allokation durch Stack-Allokation

  $\Longrightarrow$ Unterstützung der Freigabe überflüssiger Heap-Objekte
- Verringerung der Zugriffskosten

  $\Longrightarrow$ Verkürzung der Indirektionsketten (Unboxing)

1. Cache-Optimierung:

Idee: lokale Speicherzugriffe

- Laden aus dem Speicher lädt nicht nur ein Byte, sondern füllt eine ganze Cache-Zeile.
- Zugriff auf benachbarte Zellen werden billiger.
- Passen alle Daten einer inneren Schleife in den Cache, wird die Iteration extrem speicher-effizient ...

Mögliche Lösungen:

$\rightarrow$ Organisiere Zugriffe auf die vorhanden Daten um !

$\rightarrow$ Organisiere die Daten um !

Solche Optimierungen funktionieren i.a. automatisch nur für Felder :-(

Beispiel:

$$
\begin{array}{l}
\textsf{for } (j = 1; j < n; j{+}{+}) \\
\quad \textsf{for } (i = 1; i < m; i{+}{+}) \\
\qquad a[i][j] = a[i-1][j-1] + a[i][j];
\end{array}
$$

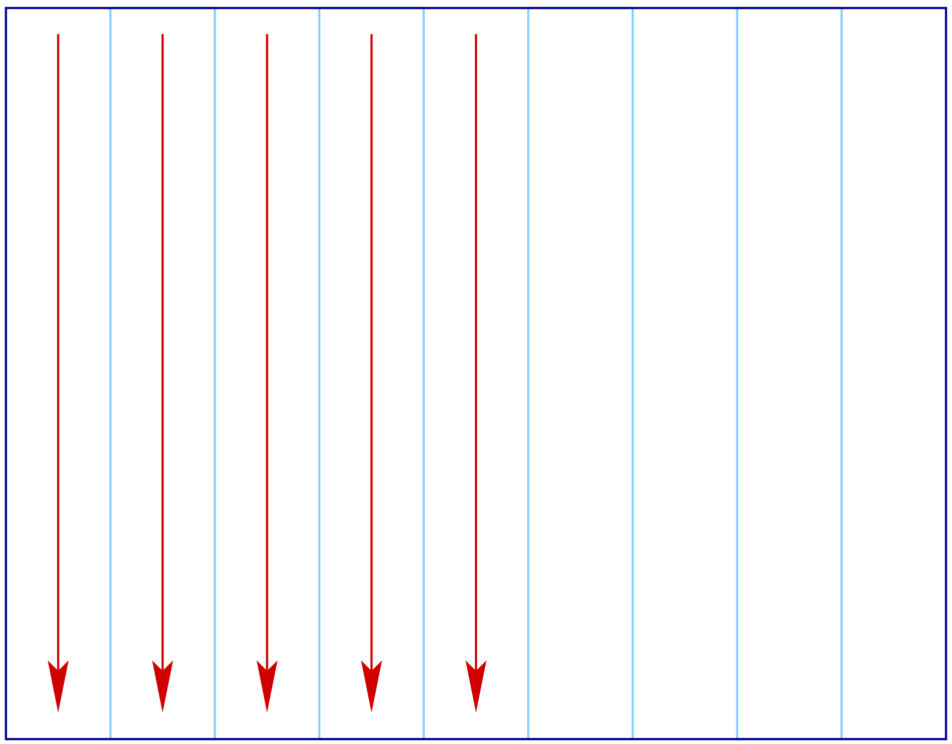
$\Longrightarrow$ Iteriere stets erst über die Zeilen!

$\Longrightarrow$ Vertausche die Reihenfolge der Iterationen:

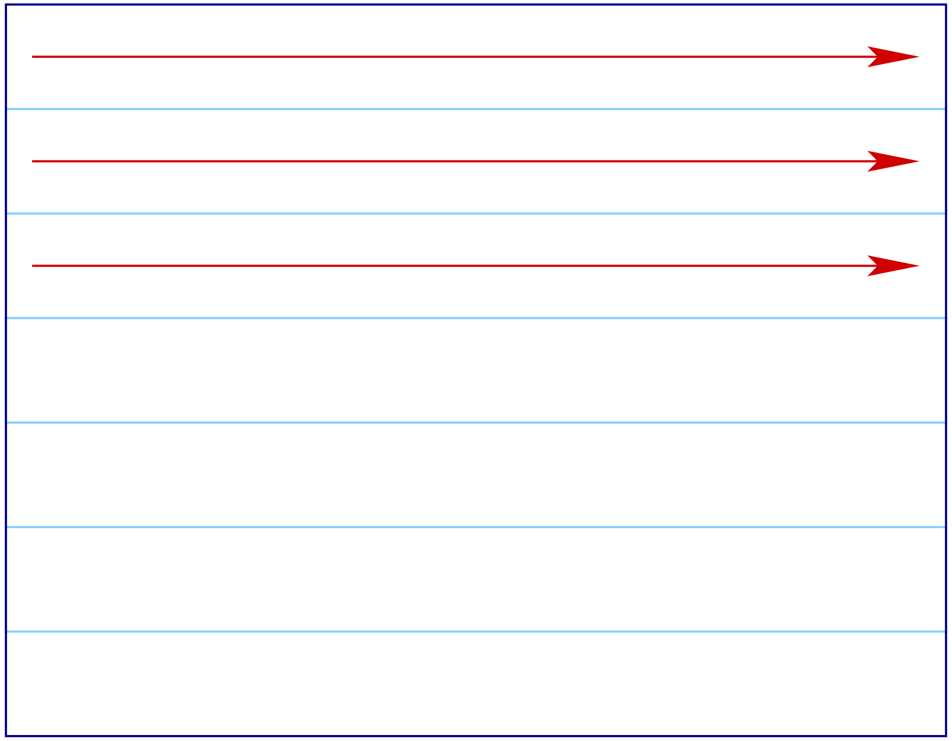
$$
\begin{array}{l}
\mathsf{for}\ \ (i = 1; i < m; i{+}{+}) \\
\quad\quad \mathsf{for}\ \ (j = 1; j < n; j{+}{+}) \\
\quad\quad\quad\quad a[i][j] = a[i-1][j-1] + a[i][j];
\end{array}
$$

Wann ist das erlaubt ???

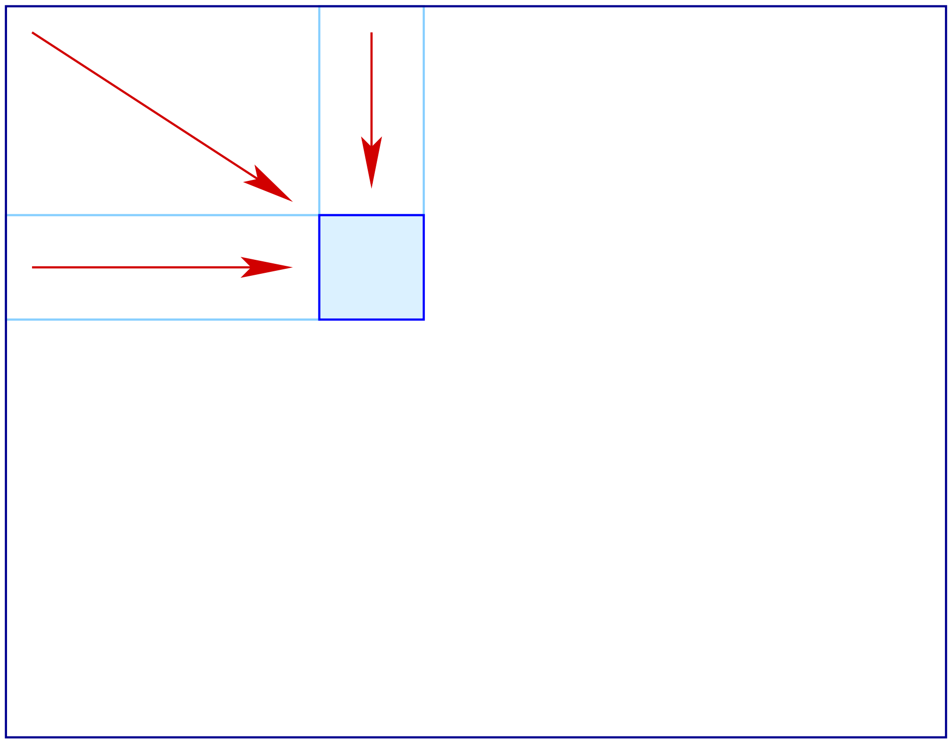
Iterations-Schema: vorher:



Iterations-Schema: nachher:



Iterations-Schema: erlaubte Abhängigkeiten:



In unserem Fall müssen wir überprüfen, dass die folgenden Gleichungs-Systeme keine Lösung haben:

Schreiben		Lesen
(i_1, j_1)	$=$	$(i_2 - 1, j_2 - 1)$
i_1	$\leq$	i_2
j_2	$\leq$	j_1
(i_1, j_1)	$=$	$(i_2 - 1, j_2 - 1)$
i_2	$\leq$	i_1
j_1	$\leq$	j_2

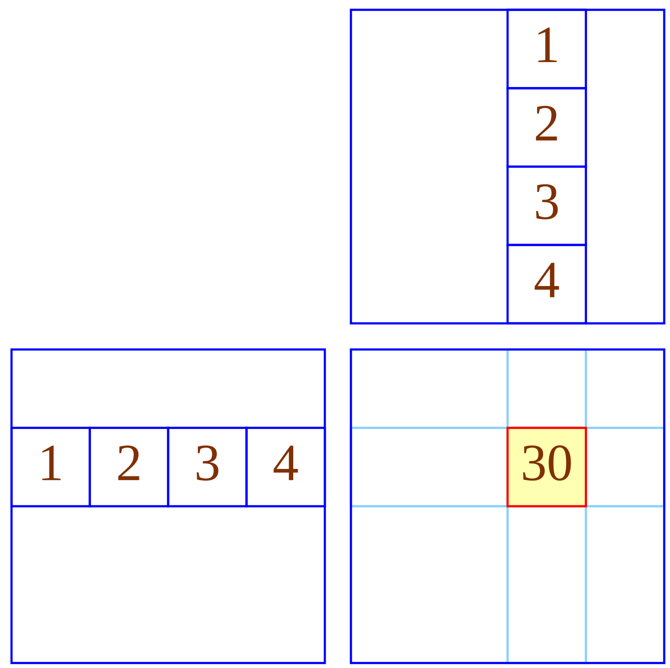
Das erste impliziert: $j_2 \leq j_2 - 1$ Hurra!

Das zweite impliziert: $i_2 \leq i_2 - 1$ Hurra!

Beispiel: Matrix-Matrix-Multiplikation

$$
\begin{array}{l}
\textsf{for}\ \ (i = 0; i < N; i{+}{+}) \\
\quad \textsf{for}\ \ (j = 0; j < M; j{+}{+}) \\
\quad\quad \textsf{for}\ \ (k = 0; k < K; k{+}{+}) \\
\quad\quad\quad c[i][j] = c[i][j] + a[i][k] \cdot b[k][j];
\end{array}
$$

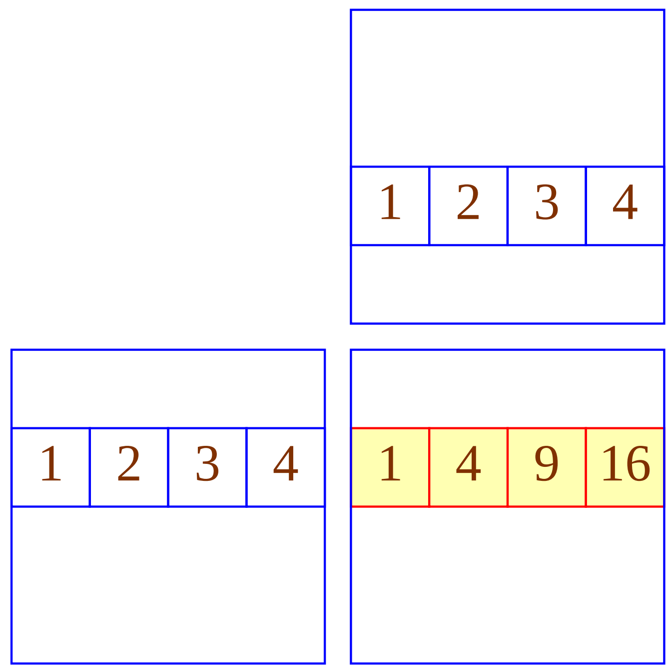
Über $b[][]$ iterieren wir spaltenweise :-(


1
2
3
4
1 2 3 4
30

Vertauschе die beiden inneren Schleifen:

$$
\begin{array}{l}
\text{for } (i = 0; i < N; i{+}{+}) \\
\qquad \text{for } (k = 0; k < K; k{+}{+}) \\
\qquad\qquad \text{for } (j = 0; j < M; j{+}{+}) \\
\qquad\qquad\qquad c[i][j] = c[i][j] + a[i][k] \cdot b[k][j];
\end{array}
$$

Ist das erlaubt ???


1 2 3 4
1 2 3 4
1 4 9 16

Diskussion:

- Die Korrektheit folgt genauso wie eben :-)
- Eine ähnliche Idee lässt sich auch zur Implementierung von Matrix-Multiplikation zeilen-komprimierter Matrizen benutzen :-))
- Möglicherweise muss das Programm erst konditioniert werden, damit die Anwendbarkeit der Transformation erkannt wird :-(
- Matrix-Multiplikation benötigt evt. erst eine Initialisierung der Ergebnis-Matrix ...

$$
\begin{array}{l}
\textsf{for}\ \ (i = 0; i < N; i{+}{+}) \\
\quad \textsf{for}\ \ (j = 0; j < M; j{+}{+})\ \ \{ \\
\quad\quad c[i][j] = 0; \\
\quad\quad \textsf{for}\ \ (k = 0; k < K; k{+}{+}) \\
\quad\quad\quad c[i][j] = c[i][j] + a[i][k] \cdot b[k][j]; \\
\quad \}
\end{array}
$$

- Jetzt können wir die beiden Iterationen nicht einfach vertauschen :-(
- Wir können aber die Iteration über j duplizieren ...

$$
\begin{array}{l}
\textsf{for}\ \ (i = 0; i < N; i{+}{+})\ \ \{ \\
\qquad \textsf{for}\ \ (j = 0; j < M; j{+}{+})\ \ c[i][j] = 0; \\
\qquad \textsf{for}\ \ (j = 0; j < M; j{+}{+}) \\
\qquad\qquad \textsf{for}\ \ (k = 0; k < K; k{+}{+}) \\
\qquad\qquad\qquad c[i][j] = c[i][j] + a[i][k] \cdot b[k][j]; \\
\}
\end{array}
$$

Zur Korrektheit:

$\Longrightarrow$ Die gelesenen Einträge (hier: keine) dürfen im Rest des Rumpfs nicht modifiziert werden !!!

$\Longrightarrow$ Die Reihenfolge der Schreibzugriffe einer Zelle darf sich nicht ändern :-)

Man erhält:

$$
\begin{array}{l}
\text{for } (i = 0; i < N; i{+}{+})\ \{ \\
\quad \text{for } (j = 0; j < M; j{+}{+})\ \ c[i][j] = 0; \\
\quad \text{for } (k = 0; k < K; k{+}{+}) \\
\quad\quad \text{for } (j = 0; j < M; j{+}{+}) \\
\quad\quad\quad c[i][j] = c[i][j] + a[i][k] \cdot b[k][j]; \\
\}
\end{array}
$$

Diskussion:

- Statt mehrere Schleifen zusammen zu fassen, haben wir Schleifen distribuiert :-)
- Desgleichen zieht man Abfragen vor die Schleife $\Longrightarrow$ if-Distribution ...

Achtung:

Statt dieser Transformation könnte man die innere Schleife auch anders optimieren:

$$
\begin{array}{l}
\textsf{for} \;\; (i = 0; i < N; i{+}{+}) \\
\qquad \textsf{for} \;\; (j = 0; j < M; j{+}{+}) \;\; \{ \\
\qquad\qquad t = 0; \\
\qquad\qquad \textsf{for} \;\; (k = 0; k < K; k{+}{+}) \\
\qquad\qquad\qquad t = t + a[i][k] \cdot b[k][j]; \\
\qquad\qquad c[i][j] = t; \\
\qquad \}
\end{array}
$$

Idee:

Finden wir ein heftig benutztes Feld-Element $a[e_1] \ldots [e_r]$, dessen Index-Ausdrücke e_l innerhalb der inneren Schleife konstant sind, können wir stattdessen ein Hilfsregister spendieren :-)

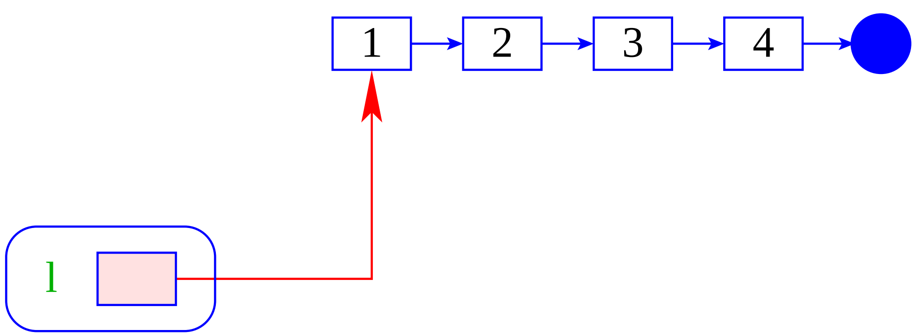
Achtung:

Diese Optimierung verhindert die vorherige und umgekehrt ...

Diskussion:

- Die bisherigen Optimierungen beziehen sich auf Iterationen über Feldern.
- Cache-sensible Organisation anderer Datenstrukturen ist möglich, aber i.a. nicht vollautomatisch möglich ...

Beispiel: Keller



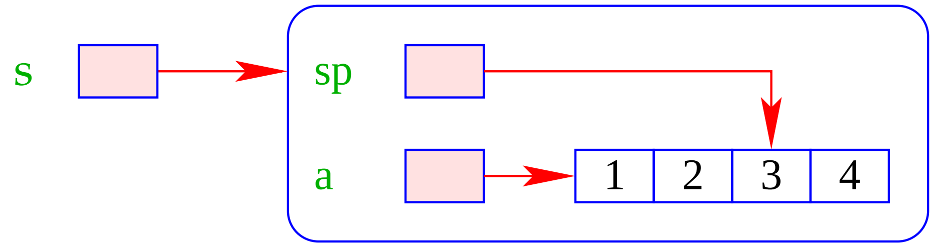
Vorteil:

- \+ Die Implementierung ist einfach :-)
- \+ Die Operationen push / pop erfordern konstante Zeit :-)
- \+ Die Datenstruktur ist potentiell beliebig groß :-)

Nachteil:

- − Die einzelnen Listenknoten können beliebig über den Speicher verteilt sein :-(

Alternative:



Vorteil:

- $+$ Die Implementierung ist auch einfach :-)
- $+$ Die Operationen push / pop erfordern konstante Zeit :-)
- $+$ Die Daten liegen konsequtiv; Stack-Schwankungen sind im Mittel gering

$\Longrightarrow$ gutes Cache-Verhalten !!!

Nachteil:

– Die Datenstruktur ist beschränkt :-(

Verbesserung:

- Ist das Feld voll, ersetze es durch ein doppelt so großes !!!
- Wird das Feld leer bis auf ein Viertel, halbiere es wieder !!!

$\Longrightarrow$ Die Extra-Kosten sind amortisiert konstant :-)

$\Longrightarrow$ Die Implementierung ist nicht mehr ganz so trivial :-}

Diskussion:

$\rightarrow$ Die gleiche Idee klappt auch für Schlangen :-)

$\rightarrow$ Andere Datenstrukturen bemüht man sich, blockweise aufzuteilen.

Problem: wie organisiert man die Zugriffe, dass sie möglichst lange auf dem selben Block arbeiten ???

$\Longrightarrow$ Algorithmen auf externen Daten

2. Stack-Allokation statt Heap-Allokation

Problem:

- Programmiersprachen wie Java legen alle Datenstrukturen im Heap an — selbst wenn sie nur innerhalb der aktuellen Methode benötigt werden :-(
- Überlebt kein Verweis auf diese Daten den Aufruf, wollen wir sie auf dem Stack allokieren :-)

$\Longrightarrow$ Escape-Analyse

Idee:

Berechne Alias-Information.

Bestimme, ob ein erzeugtes Objekt möglicherweise von außen erreichbar ist ...

Beispiel: unsere Pointer-Sprache

$$x = \mathsf{new}();$$
$$y = \mathsf{new}();$$
$$x \to a = y;$$
$$z = y;$$
$$\mathsf{return}\ \ z;$$

... könnte ein möglicher Methoden-Rumpf sein ;-)

Von außen zugänglich sind Objekte, die:

- von return zurück geliefert werden;
- einer globalen Variablen zugewiesen werden;
- von solchen Objekten erreichbar sind.

... im Beispiel:

$$x = \mathsf{new}();$$
$$y = \mathsf{new}();$$
$$x \rightarrow a = y;$$
$$z = y;$$
$$\mathsf{return}\ \boxed{z};$$

Von außen zugänglich sind Objekte, die:

- von `return` zurück geliefert werden;
- einer globalen Variablen zugewiesen werden;
- von solchen Objekten erreichbar sind.

... im Beispiel:

$$
\begin{array}{l}
x = \mathsf{new}();\\
y = \mathsf{new}();\\
x \longrightarrow a = y;\\
z = \boxed{y};\\
\mathsf{return}\ \boxed{z};
\end{array}
$$

Von außen zugänglich sind Objekte, die:

- von return zurück geliefert werden;
- einer globalen Variablen zugewiesen werden;
- von solchen Objekten erreichbar sind.

... im Beispiel:

$$x = \mathsf{new}();$$

$$y = \mathsf{new}();$$

$$\boxed{x \longrightarrow a} = y;$$

$$z = \boxed{y};$$

$$\mathsf{return}\ \boxed{z};$$

Von außen zugänglich sind Objekte, die:

- von `return` zurück geliefert werden;
- einer globalen Variablen zugewiesen werden;
- von solchen Objekten erreichbar sind.

... im Beispiel:

$$
\begin{array}{l}
x = \mathsf{new}();\\
y = \boxed{\mathsf{new}()};\\
\boxed{x \longrightarrow a} = y;\\
z = \boxed{y};\\
\mathsf{return}\ \ \boxed{z};
\end{array}
$$

Wir schließen:

- Die Objekte, die das erste `new()` anlegt, können nicht entkommen.
- Wir können sie darum auf dem Stack allokieren :-)

Achtung:

Das ist natürlich nur sinnvoll, wenn von dieser Sorte nur wenige pro Methoden-Aufruf angelegt werden :-(

Liegt deshalb ein solches lokales `new()` in einer Schleife, sollten wir die Objekte vorsichtshalber doch im Heap anlegen ;-)

Erweiterung: Formale Parameter

- Wir benötigen eine interprozedurale Alias-Analyse :-)
- Kennen wir das gesamte Programm, können wir z.B. die Kontrollflussgraphen der einzelnen Funktionen zu einem einzigen zusammen fassen (durch Hinzufügen geeigneter Kanten) und für diesen Alias-Information berechnen ...
- Achtung: benutzen wir die selben Namen $y_1, y_2, \ldots$ für die formalen Parameter, wird die Information dort notwendig ungenau :-(
- Kennen wir das Gesamtprogramm nicht, müssen wir annehmen, dass jede Referenz, die einer anderen Funktion bekannt ist, entkommt :-((

3.5 Zusammenfassung

Wir haben jetzt diverse Optimierungen kennen gelernt zur besseren Ausnutzung der Hardware-Gegebenheiten.

Reihenfolge ihrer Anwendung:

- Erst globale Restrukturierungen der Prozeduren/Funktionen sowie der Schleifen für besseres Speicherverhalten ;-)
- Dann lokale Umstrukturierung für optimale Nutzung des Instruktionssatzes und der Prozessor-Parallelität :-)
- Dann Registerverteilung und schließlich
- Peephole-Optimierung für den letzten Schliff ...

Funktionen:	Endrekursion + Inlining Stack-Allokation
Schleifen:	Iterationsverbesserung $\longrightarrow$ if-Distribution $\longrightarrow$ for-Distribution Werte-Caching
Rümpfe:	Life-Range-Splitting Instruktions-Auswahl Instruktions-Anordung mit $\longrightarrow$ Schleifen-Abwicklung $\longrightarrow$ Schleifen-Verschmelzung
Instruktionen:	Register-Verteilung Peephole-Optimierung

4 Optimierung funktionaler Programme

Beispiel:

$$\begin{array}{lll} \textbf{fun}\ \ \mathsf{fac}\ x & = & \textbf{if}\ \ x \leq 1\ \ \textbf{then}\ \ 1 \\ & & \textbf{else}\ \ x \cdot \mathsf{fac}\ (x-1) \end{array}$$

- Es gibt keine Basis-Blöcke :-(
- Es gibt keine Schleifen :-(
- Viele Funktionen sind rekursiv :-((

Strategien zur Optimierung:

$\Longrightarrow$ Verbessere spezielle Ineffizienzen wie:

- Pattern Matching
- Lazy Evaluation (falls vorhanden ;-)
- Indirektionen — Unboxing / Escape-Analyse
- Zwischendatenstrukturen — Deforestation

$\Longrightarrow$ Entdecke bzw. erzeuge Schleifen mit Basis-Blöcken :-)

- Endrekursion
- Inlining
- **let**-Floating

Wende dann allgemeine Optimierungs-Techniken an!

... etwa durch Übersetzung nach C ;-)

Achtung:

Wir benötigen neue Programmanalyse-Techniken, um Informationen über funktionale Programme zu sammeln.

Beispiel: Inlining

$$\begin{array}{llll} \textbf{fun} & \mathsf{max}\,(x, y) & = & \textbf{if}\ \ x > y\ \ \textbf{then}\ \ x \\ & & & \textbf{else}\ \ y \\ \textbf{fun} & \mathsf{abs}\ z & = & \mathsf{max}\,(z, -z) \end{array}$$

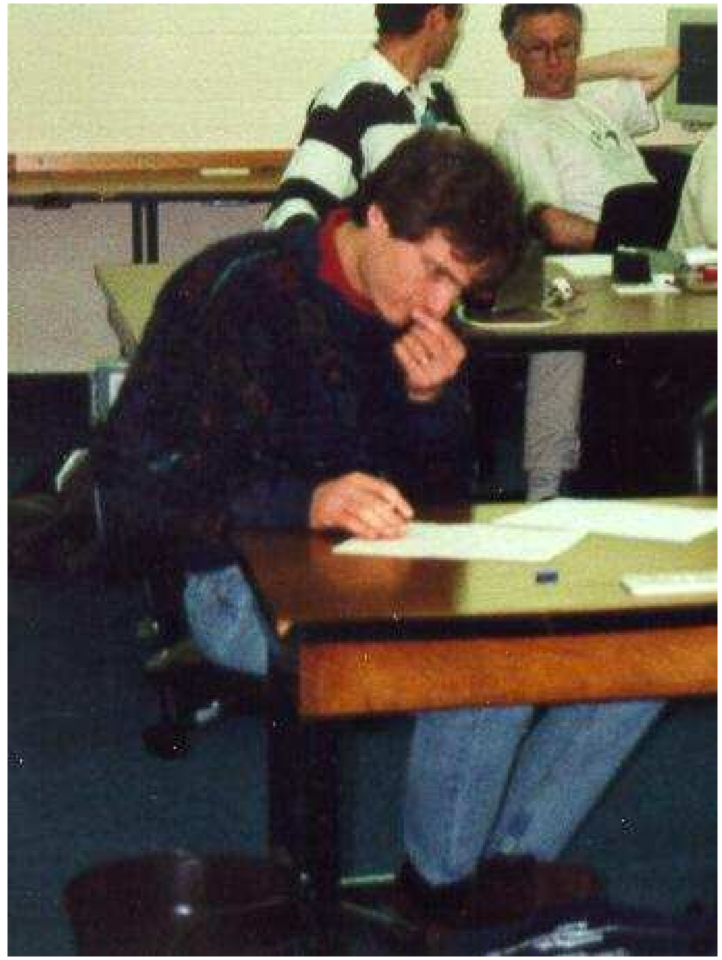
Als Ergebnis der Optimierung erwarten wir ...

$$
\begin{array}{lcll}
\textbf{fun}\ \ \textsf{max}\,(x, y) & = & \textbf{if}\ \ x > y\ \ \textbf{then}\ \ x & \\
 & & \textbf{else}\ \ y & \\
\textbf{fun}\ \ \textsf{abs}\ z & = & \textbf{let} & \textbf{val}\ \ x = z \\
 & & & \textbf{val}\ \ y = -z \\
 & & \textbf{in} & \boxed{\begin{array}{l}\textbf{if}\ \ x > y\ \ \textbf{then}\ \ x \\ \textbf{else}\ \ y\end{array}} \\
 & & \textbf{end} &
\end{array}
$$

Diskussion:

max ist zuerstmal nur ein Name. Wirt müssen herausfinden, welchen Wert er zur Laufzeit haben kann

$\Longrightarrow$ Wert-Analyse erforderlich !!



Nevin Heintze im australischen Team
des Prolog-Programmier-Wettbewerbs, 1998

Das ganze Bild:

4.1 Eine einfache Zwischensprache

Zur Vereinfachung betrachten wir:

$$
\begin{array}{lcl}
v & ::= & b \mid (x_1, \ldots, x_k) \mid c\ x \mid \textbf{fn}\ x \Rightarrow e \\
e & ::= & v \mid (x_1\ x_2) \mid (\Box_1\ x) \mid (x_1\ \Box_2\ x_2) \mid \\
 & & \textbf{let}\ x_1 = e_1\ \ldots\ x_k = e_k\ \textbf{in}\ e_0\ \textbf{end} \mid \\
 & & \textbf{letrec}\ x_1 = e_1\ \ldots\ x_k = e_k\ \textbf{in}\ e_0\ \textbf{end} \mid \\
 & & \textbf{case}\ x\ \textbf{of}\ p_1 :\ e_1\ \mid \ldots \mid\ p_k :\ e_k\ \textbf{end} \\
p & ::= & v \mid x \mid c\ x \mid (x_1, \ldots, x_k)
\end{array}
$$

wobei b eine Konstante ist, x eine Variable, c ein (Daten-)Konstruktor und $\Box_i$ i-stellige Operatoren sind.

Diskussion:

- Konstruktoren und Funktionen sind stets ein-stellig.
  Dafür gibt es explizite Tupel :-)
- **if**-Ausdrücke und Fall-Unterscheidung in Funktions-Definitionen wird auf **case**-Ausdrücke zurückgeführt.
- In Fall-Unterscheidungen sind nur einfache Muster erlaubt.

  $\Longrightarrow$ Komplizierte Muster müssen zerlegt werden ...
- **let**-Definitionen entsprechen Basis-Blöcken :-)
- Typ-Annotationen an Variablen, Mustern oder Ausdrücken könnten weitere nützliche Informationen enthalten
  — wir verzichten aber drauf :-)

... im Beispiel:

Die Definition von max sieht dann so aus:

$$
\begin{array}{lll}
\mathsf{max} & = & \textbf{fn}\ x \Rightarrow\ \ \textbf{case}\ x\ \textbf{of}\ (x_1, x_2)\ :\\
& & \qquad\qquad \textbf{let}\ z = x_1 < x_2\\
& & \qquad\qquad \textbf{in}\ \ \textbf{case}\ z\\
& & \qquad\qquad\qquad \textbf{of}\ \ \mathsf{True}\ :\ \ x_2\\
& & \qquad\qquad\qquad\ \ \mid\ \ \mathsf{False}\ :\ \ x_1\\
& & \qquad\qquad\qquad \textbf{end}\\
& & \qquad\qquad \textbf{end}\\
& & \qquad \textbf{end}
\end{array}
$$

Entsprechend haben wir für abs :

$$\begin{aligned} \mathsf{abs} \;=\; \mathbf{fn}\; x \Rightarrow \;\; & \mathbf{let}\; z_1 = -x \\ & \qquad z_2 = (x, z_1) \\ & \mathbf{in}\; (\mathsf{max}\; z_2) \\ & \mathbf{end} \end{aligned}$$

4.2 Eine einfache Wert-Analyse

Idee:

Für jeden Teilausdruck e sammeln wir die Menge $[\![e]\!]^\sharp$ der möglichen Werte von e ...

Sei V die Menge der vorkommenden Konstanten (-Klassen), Konstruktor-Anwendungen und Funktionen. Dann wählen wir als vollständigen Verband natürlich:

$$\mathbb{V} = 2^V$$

Wir stellen wir ein Ungleichungs-System auf:

- Ist e ein Wert d.h. von der Form: $b, c\, x, (x_1, \ldots, x_k)$ oder $\textbf{fn}\, x \Rightarrow e$ erzeugen wir:

$$[\![e]\!]^\sharp \supseteq \{e\}$$

- Ist $e \equiv (x_1\; x_2)$ und $f \equiv \textbf{fn}\; x \Rightarrow e_1$, dann

$$\begin{array}{lll} [\![e]\!]^\sharp & \supseteq & (f \in [\![x_1]\!]^\sharp)\, ?\, [\![e_1]\!]^\sharp \;:\; \emptyset \\ [\![x]\!]^\sharp & \supseteq & (f \in [\![x_1]\!]^\sharp)\, ?\, [\![x_2]\!]^\sharp \;:\; \emptyset \end{array}$$

...

- int-Werte, die Operatoren zurück liefern, approximieren wir z.B. durch eine Konstante int.

  Operatoren, die Boolesche Werte liefern, liefern z.B. $\{\mathsf{True}, \mathsf{False}\}$:-)

- Ist $e \equiv \textbf{let } x_1 = e_1 \ldots x_k = e_k \textbf{ in } e_0 \textbf{ end}$. Dann erzeugen wir:

$$\begin{array}{lll} [\![x_i]\!]^\sharp & \supseteq & [\![e_i]\!]^\sharp \\ [\![e]\!]^\sharp & \supseteq & [\![e_0]\!]^\sharp \end{array}$$

- Analog für $e \equiv \textbf{letrec } x_1 = e_1 \ldots x_k = e_k \textbf{ in } e_0 \textbf{ end}$:

$$\begin{array}{lll} [\![x_i]\!]^\sharp & \supseteq & [\![e_i]\!]^\sharp \\ [\![e]\!]^\sharp & \supseteq & [\![e_0]\!]^\sharp \end{array}$$

- Sei $e \equiv \textbf{case } x \textbf{ of } p_1 \;:\; e_1 \mid \ldots \mid p_k \;:\; e_k \textbf{ end}$.
  Dann erzeugen wir für $p_i \equiv b$,

$$[\![e]\!]^\sharp \supseteq (b \in [\![x]\!]^\sharp) \,?\, [\![e_i]\!]^\sharp \;:\; \emptyset$$

Ist $p_i \equiv c\, y$ und $v \equiv c\, z$ ein Wert, dann

$$\begin{array}{lll} [\![e]\!]^\sharp & \supseteq & (v \in [\![x]\!]^\sharp) \,?\, [\![e_i]\!]^\sharp \;:\; \emptyset \\ [\![y]\!]^\sharp & \supseteq & (v \in [\![x]\!]^\sharp) \,?\, [\![z]\!]^\sharp \;:\; \emptyset \end{array}$$

Ist $p_i \equiv (y_1, \ldots, y_k)$ und $v \equiv (z_1, \ldots, z_k)$ ein Wert, dann

$$\begin{array}{lll} [\![e]\!]^\sharp & \supseteq & (v \in [\![x]\!]^\sharp) \,?\, [\![e_i]\!]^\sharp \;:\; \emptyset \\ [\![y_j]\!]^\sharp & \supseteq & (v \in [\![x]\!]^\sharp) \,?\, [\![z_j]\!]^\sharp \;:\; \emptyset \end{array}$$

Ist $p_i \equiv y$, dann

$$\begin{array}{lll} [\![e]\!]^\sharp & \supseteq & [\![e_i]\!]^\sharp \\ [\![y]\!]^\sharp & \supseteq & [\![x]\!]^\sharp \end{array}$$

4.3 Eine operationelle Semantik

Idee:

Wir konstruieren eine Big-Step operationelle Semantik, die Ausdrücke auswertet :-)

Konfigurationen:

$$
\begin{array}{lll}
c & ::= & (e, env) \\
vc & ::= & (v, env) \\
env & ::= & \{x_1 \mapsto vc_1, \ldots\}
\end{array}
$$

Werte sind Konfigurationen, in denen der Ausdruck von der Form:
$b, c\, x, (x_1, \ldots, x_k)$ oder $\mathbf{fn}\, x \Rightarrow e$ ist :-)

Umgebungen enthalten nur Werte :-))

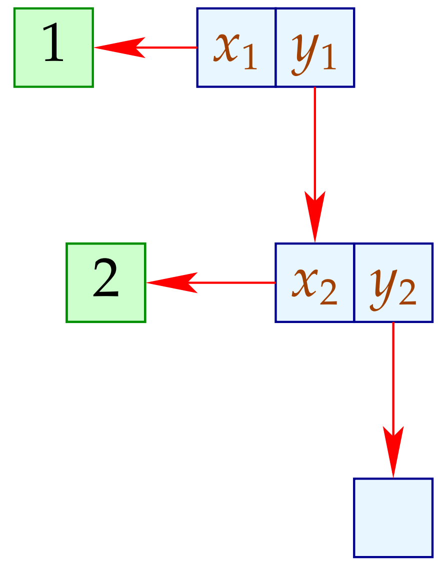
Beispiele für Werte:

$$\begin{array}{lll}
1 & : & (1, \emptyset) \\
c\,1 & : & (c\ x, \{x \mapsto (1, \emptyset)\}) \\
[1, 2] & : & ((x_1, y_1), \{x_1 \mapsto 1, \\
& & \quad y_1 \mapsto ((x_2, y_2), \{x_2 \mapsto 2, \\
& & \qquad y_2 \mapsto ((), \emptyset)\})\})
\end{array}$$

Werte sehen etwas merkwürdig aus :-)

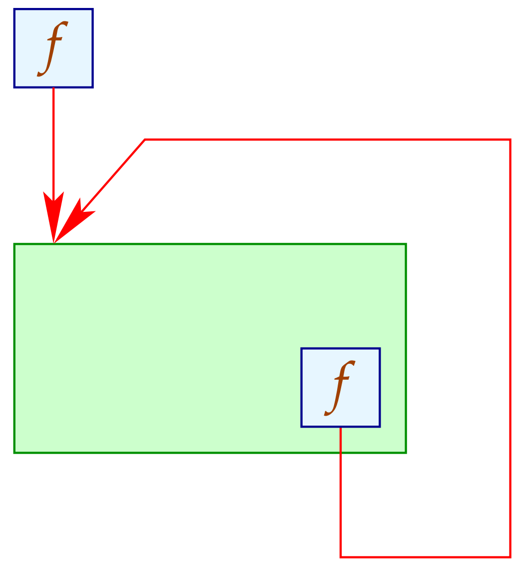
Der Grund ist, dass wir Substitutionen nie ausführen :-)

Alternativ können wir uns die Variablen in den Umgebungen als Speicherzellen vorstellen ...



Achtung:

Rekursive Funktionen führen zu zyklischen Verweis-Strukturen
;-)



Auswege:

- Rekursive Funktionen werden auf dem Toplevel definiert :-)
- Lokale Rekursive Funktionen sind stets nur selbst rekursiv.

  Für diese führen wir einen neuen Operator fix ein ...

Aus: $\quad$ $\textbf{letrec } x_1 = e_1 \textbf{ in } e_0 \textbf{ end}$

wird: $\quad$ $\textbf{let } x_1 = \mathsf{fix}\,(x_1, e_1) \textbf{ in } e_0 \textbf{ end}$

Beispiel: Die append-Funktion

Betrachten wir die Konkatenation von zwei Listen. In ML schreiben wir einfach:

$$
\begin{array}{lllcl}
\textbf{fun} & \mathsf{app} & [\,] & = & \textbf{fn } y \Rightarrow y \\
\mid & \mathsf{app} & (x :: xs) & = & \textbf{fn } y \Rightarrow x :: \mathsf{app}\ xs\ y
\end{array}
$$

In unserer eingeschränkten Zwischensprache sieht das etwas detaillierter aus :-)

$$
\begin{array}{lcl}
\mathsf{app} & = & \mathsf{fix}\,(\mathsf{app},\ \mathbf{fn}\ x \Rightarrow \mathbf{case}\ x \\
& & \qquad\qquad \mathbf{of}\ \ [\,]\ :\ \ \mathbf{fn}\ y\ \Rightarrow\ y \\
& & \qquad\qquad \mid\ \ ::\ z :\ \ \mathbf{case}\ z\ \mathbf{of}\ (x_1, x_2)\ :\ \ \mathbf{fn}\ y\ \Rightarrow \\
& & \qquad\qquad\qquad\qquad \mathbf{let}\ \ a_1\ =\ \mathsf{app}\ x_2 \\
& & \qquad\qquad\qquad\qquad\qquad\ a_2\ =\ a_1\ y \\
& & \qquad\qquad\qquad\qquad\qquad\ z_1\ =\ (x_1, a_2) \\
& & \qquad\qquad\qquad\qquad \mathbf{in}\ \ ::\ z_1 \\
& & \qquad\qquad\qquad\qquad \mathbf{end} \\
& & \qquad\qquad\qquad \mathbf{end} \\
& & \qquad\qquad \mathbf{end}\ )
\end{array}
$$

Die Big-Step Semantik gibt Regeln an, zu welchem Wert sich eine Konfiguration ausrechnen lässt ...

Funktionsanwendung:

$$\dfrac{\begin{array}{l} \eta\, x_1 \;=\; (\mathbf{fn}\; x \;\Rightarrow\; e, \eta_1) \\ \eta\, x_2 \;=\; (v_2, \eta_2) \\ (e, \eta_1 \oplus \{x \mapsto (v_2, \eta_2)\}) \Longrightarrow (v_3, \eta_3) \end{array}}{(x_1\; x_2, \eta) \Longrightarrow (v_3, \eta_3)}$$

Lokal rekursive Funktionsanwendung:

$$\frac{\begin{array}{lll} \eta\, x_1 & = & (\mathsf{fix}\,(f, \mathbf{fn}\ x\ \Rightarrow\ e), \eta_1) \\ \eta\, x_2 & = & (v_2, \eta_2) \\ \multicolumn{3}{l}{(e, \eta_1 \oplus \{f \mapsto (\mathsf{fix}\,(f, \mathbf{fn}\ x\ \Rightarrow\ e), \eta_1),\ x \mapsto (v_2, \eta_2)\}) \Longrightarrow (v_3, \eta_3)} \end{array}}{(x_1\ x_2, \eta) \Longrightarrow (v_3, \eta_3)}$$

Fall-Unterscheidung 1:

$$\frac{\begin{array}{l} \eta\, x \;=\; (b, \eta_1) \\ (e_i, \eta) \Longrightarrow (v_i, \eta_i) \end{array}}{(\textbf{case}\ x\ \textbf{of}\ p_1 \;:\; e_1 \mid \ldots \mid p_k \;:\; e_k\ \textbf{end}, \eta) \Longrightarrow (v_i, \eta_i)}$$

sofern $p_i \equiv b$ das erste auf b passende Muster ist :-)

Fall-Unterscheidung 2:

$$\frac{\begin{array}{l} \eta\, x \;=\; (c\, z, \eta_1) \\ (e_i, \eta \oplus \{x_i \mapsto (\eta\, z)\}) \Longrightarrow (v_i, \eta_i) \end{array}}{(\textbf{case}\ x\ \textbf{of}\ p_1\ :\ e_1 \mid \ldots \mid p_k\ :\ e_k\ \textbf{end}, \eta) \Longrightarrow (v_i, \eta_i)}$$

sofern $p_i \equiv c\ x_i$ das erste auf $c\, z$ passende Muster ist :-)

Fall-Unterscheidung 3:

$$\frac{\begin{array}{l} \eta\, x \;=\; ((z_1, \ldots, z_m), \eta_1) \\ (e_i, \eta \oplus \{y_j \mapsto (\eta\, z_j) \mid j = 1, \ldots, m\}) \Longrightarrow (v_i, \eta_i) \end{array}}{(\mathbf{case}\; x\; \mathbf{of}\; p_1 \;:\; e_1 \mid \ldots \mid p_k \;:\; e_k\; \mathbf{end}, \eta) \Longrightarrow (v_i, \eta_i)}$$

für das erste passende Muster $\quad p_i \equiv (y_1, \ldots, y_m) \quad$:-)

Fall-Unterscheidung 4:

$$\frac{(e_i, \eta \oplus \{x_i \mapsto (\eta\, x)\}) \Longrightarrow (v_i, \eta_i)}{(\textbf{case}\ x\ \textbf{of}\ p_1\ :\ e_1 \mid \ldots \mid p_k\ :\ e_k\ \textbf{end}, \eta) \Longrightarrow (v_i, \eta_i)}$$

sofern $p_i \equiv x_i$ und alle Muster davor fehl schlugen :-)

Lokale Definitionen:

$$
\frac{\begin{array}{l}
(e_1, \eta) \Longrightarrow (v_1, \eta_1) \\
(e_2, \eta \oplus \{x_1 \mapsto (v_1, \eta_1)\}) \Longrightarrow (v_2, \eta_2) \\
\ldots \\
(e_k, \eta \oplus \{x_1 \mapsto (v_1, \eta_1), \ldots, x_{k-1} \mapsto (v_{k-1}, \eta_{k-1})\}) \Longrightarrow (v_k, \eta_k) \\
(e_0, \eta \oplus \{x_1 \mapsto (v_1, \eta_1), \ldots, x_k \mapsto (v_k, \eta_k)\}) \Longrightarrow (v_0, \eta_0)
\end{array}}{(\mathbf{let}\ x_1 = e_1 \ldots x_k = e_k\ \mathbf{in}\ e_0\ \mathbf{end}, \eta) \Longrightarrow (v_0, \eta_0)}
$$

Variablen:

$$\frac{\eta\ (x)\ =\ (v_1, \eta_1)}{(x, \eta) \Longrightarrow (v_1, \eta_1)}$$

Korrektheit der Analyse:

Man zeigt für jedes (e, η) , das in einer Ableitung für das Programm vorkommt:

- Falls $\eta\,(x) = (v, \eta_1)$, dann ist $v \in [\![x]\!]^\sharp$.
- Falls $(e, \eta) \Longrightarrow (v, \eta_1)$, dann ist $v \in [\![e]\!]^\sharp$.

Fazit:

$[\![e]\!]^\sharp$ liefert eine Obermenge der Werte, zu denen sich e möglicherweise ausrechnet :-)

4.4 Anwendung: Inlining

Probleme:

- globale Variablen. Das Programm:

$$\begin{array}{lllll} \textbf{let} & x = 1 & & & \\ & f = & \textbf{let} & x = 2 & \\ & & \textbf{in} & \textbf{fn}\ y \Rightarrow y + x & \\ & & \textbf{end} & & \\ \textbf{in} & f\ x & & & \\ \textbf{end} & & & & \end{array}$$

... berechnet offenbar etwas anderes als:

$$
\begin{array}{lll}
\textbf{let} & x = 1 & \\
 & f = & \begin{array}{ll} \textbf{let} & x = 2 \\ \textbf{in} & \textbf{fn}\ y \Rightarrow y + x \\ \textbf{end} & \end{array} \\
\textbf{in} & \boxed{\begin{array}{ll} \textbf{let} & y = x \\ \textbf{in} & y + x \\ \textbf{end} & \end{array}} & \\
\textbf{end} & &
\end{array}
$$

- rekursive Funktionen. In der Definition:

$$x = \mathsf{fix}\ (\mathsf{foo},\ \textbf{fn}\ y \Rightarrow \mathsf{foo}\ y)$$

sollten wir foo besser nicht substituieren :-)

Idee 1:

→ Wir machen erstmal die Namen im Programm eindeutig.

→ Dann substituieren wir nur Funktionen, die statisch im Scope der selben globalen Variablen stehn, wie die Anwendung :-)

→ Wir berechnen für jeden Ausdruck alle Funktions-Definitionen mit dieser Eigenschaft :-)

Sei $D[e]$ die Menge der Definitionen, die in e statisch ankommen.

- Für $e \equiv \textbf{let}\ x_1 = e_1\ \ldots\ x_k = e_k\ \textbf{in}\ e_0\ \textbf{end}$ haben wir:

$$\begin{aligned} D[e_1] &= D \\ &\ldots \\ D[e_k] &= D \cup \{x_1, \ldots, x_{k-1}\} \\ D[e_0] &= D \cup \{x_1, \ldots, x_k\} \end{aligned}$$

- In den anderen Fällen propagiert sich D unverändert zu den Teilausdrücken :-)

  Für $e \equiv \textbf{fn}\ x \Rightarrow e_1$ haben wir etwa:

$$D[e_1] = D$$

... im Beispiel:

$$
\begin{array}{lll}
\textbf{let} & x = 1 & \\
 & f = \begin{array}{ll} \textbf{let} & x_1 = 2 \\ \textbf{in} & \textbf{fn}\ y \Rightarrow y + x_1 \\ \textbf{end} & \end{array} & \\
\textbf{in} & f\ x & \\
\textbf{end} & &
\end{array}
$$

... steht (nach Umbenennung :-) f für $f\ x$ statisch zur Verfügung. Bei Substitution erhalten wir:

$$
\begin{array}{ll}
\textbf{let} & x = 1 \\
 & f = \begin{array}[t]{ll} \textbf{let} & x_1 = 2 \\ \textbf{in} & \textbf{fn}\ y \Rightarrow y + x_1 \\ \textbf{end} & \end{array} \\
\textbf{in} & \boxed{\begin{array}{ll} \textbf{let} & y = x \\ \textbf{in} & \begin{array}[t]{ll} \textbf{let} & x_1 = 2 \\ \textbf{in} & y + x_1 \\ \textbf{end} & \end{array} \\ \textbf{end} & \end{array}} \\
\textbf{end} &
\end{array}
$$

Ersetzen der Variablen-Variablen-Umbenennungen ergibt schließlich:

$$
\begin{array}{lll}
\textbf{let} & x = 1 & \\
 & f = & \begin{array}{ll} \textbf{let} & x_1 = 2 \\ \textbf{in} & \textbf{fn}\ y \Rightarrow y + x_1 \\ \textbf{end} & \end{array} \\
\textbf{in} & \boxed{\begin{array}{ll} \textbf{let} & x_1 = 2 \\ \textbf{in} & x + x_1 \\ \textbf{end} & \end{array}} & \\
\textbf{end} & &
\end{array}
$$

Idee 2:

- $\rightarrow$ Wir benutzen unsere Wert-Analyse.
- $\rightarrow$ Wir ignorieren globale Variablen :-)
- $\rightarrow$ Wir substituieren nur Funktionen ohne freie Variablen :-))

Beispiel: Die map-Funktion

$$
\begin{array}{l}
\textbf{let } f = \textbf{fn } x \Rightarrow x \cdot x \\
\quad \textsf{map} = \textsf{fix}\,(\textsf{map},\ \textbf{fn } g \Rightarrow \textbf{fn } x \Rightarrow \textbf{case } x \\
\qquad\qquad\qquad\qquad \textbf{of } [\,] : \ [\,] \\
\qquad\qquad\qquad\qquad\ \mid\ \ :: z : \ \textbf{case } z \textbf{ of } (x_1, x_2) \textbf{ in} \\
\qquad\qquad\qquad\qquad\qquad\qquad \textbf{let } y_1 = g\, x_1 \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad m = \textsf{map}\ g \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad y_2 = m\ x_2 \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad z_1 = (y_1, y_2) \\
\qquad\qquad\qquad\qquad\qquad\qquad \textbf{in } :: z_1 \\
\qquad\qquad\qquad\qquad\qquad\qquad \textbf{end} \\
\qquad\qquad\qquad\qquad \textbf{end}) \\
\quad h = \textsf{map}\ \ f \\
\textbf{in } \ h\ \mathit{list} \\
\textbf{end}
\end{array}
$$

- Der formale Parameter g von map ist stets f :-)
- Wir können die Anwendung von f in der Definition von map ersetzen:

$$
\begin{aligned}
\mathsf{map} = \mathsf{fix}\,(\mathsf{map},\ &\mathbf{fn}\ g\ \Rightarrow\ \mathbf{fn}\ x\ \Rightarrow\ \mathbf{case}\ x\\
&\quad \mathbf{of}\ \ [\,]\ :\ \ [\,]\\
&\quad \mid\ \ ::z:\ \ \mathbf{case}\ z\ \mathbf{of}\ (x_1, x_2)\ \mathbf{in}\\
&\qquad\qquad \mathbf{let}\ y_1 = \boxed{\begin{array}{l}\mathbf{let}\ x = x_1\\ \mathbf{in}\ x \cdot x\\ \mathbf{end}\end{array}}\\
&\qquad\qquad\quad m = \mathsf{map}\ g\\
&\qquad\qquad\quad y_2 = m\ x_2\\
&\qquad\qquad\quad z_1 = (y_1, y_2)\\
&\qquad\qquad \mathbf{in}\ \ ::z_1\\
&\qquad\qquad \mathbf{end}\\
&\quad \mathbf{end})\\
h = \mathsf{map}\ \ f &
\end{aligned}
$$

- Noch mehr könnten wir sparen, wenn wir die spezialisierte Funktion $h = \mathsf{map}\ f$ direkt definieren könnten :-)
- Dazu müssen wir überall in der Definition von map das Muster $\boxed{\mathsf{map}\ g}$ durch h ersetzen ...

$$\Longrightarrow \quad \text{fold-Transformation} \quad \text{:-)}$$

- Alle weiteren Vorkommen von g müssen durch (die Definition von) f ersetzt werden ...

  // kommt hier nicht vor :-)

$$
\begin{aligned}
\mathsf{map} = \mathsf{fix}\,(\boxed{\mathsf{map},\ \mathbf{fn}\ g\ \Rightarrow}\ \mathbf{fn}\ x\ \Rightarrow\ \mathbf{case}\ x \\
\mathbf{of}\ \ [\,]\ :\ \ [\,] \\
\mid\ \ ::z:\ \ \mathbf{case}\ z\ \mathbf{of}\ (x_1, x_2)\ \mathbf{in} \\
\mathbf{let}\ y_1 = \boxed{\begin{array}{l}\mathbf{let}\ x = x_1 \\ \mathbf{in}\ x \cdot x \\ \mathbf{end}\end{array}} \\
m = \boxed{\mathsf{map}\ g} \\
y_2 = m\ x_2 \\
z_1 = (y_1, y_2) \\
\mathbf{in}\ \ ::z_1 \\
\mathbf{end} \\
\mathbf{end})
\end{aligned}
$$

$$h = \mathsf{map}\ \ f$$

$$
\begin{array}{l}
h = \mathsf{fix}\,(h, \mathbf{fn}\ x \ \Rightarrow\ \mathbf{case}\ x \\
\qquad\qquad \mathbf{of}\ \ [\,]\ :\ \ [\,] \\
\qquad\qquad \ \ |\ \ \ ::z:\ \ \mathbf{case}\ z\ \mathbf{of}\ (x_1, x_2)\ \mathbf{in} \\
\qquad\qquad\qquad\qquad \mathbf{let}\ y_1 = \boxed{\begin{array}{l}\mathbf{let}\ x = x_1 \\ \mathbf{in}\ x \cdot x \\ \mathbf{end}\end{array}} \\
\qquad\qquad\qquad\qquad\qquad m = \boxed{h} \\
\qquad\qquad\qquad\qquad\qquad y_2 = m\ x_2 \\
\qquad\qquad\qquad\qquad\qquad z_1 = (y_1, y_2) \\
\qquad\qquad\qquad\qquad \mathbf{in}\ ::z_1 \\
\qquad\qquad\qquad\qquad \mathbf{end} \\
\qquad\qquad \mathbf{end})
\end{array}
$$

Beseitigung von Variablen-Variablen-Umspeicherungen liefert:

$$
\begin{array}{l}
h = \mathsf{fix}\,(h, \textbf{fn}\ x \ \Rightarrow\ \textbf{case}\ x \\
\qquad\qquad\qquad \begin{array}{lll}
\textbf{of} & [\,] : & [\,] \\
\mid & ::\, z : & \textbf{case}\ z\ \textbf{of}\ (x_1, x_2)\ \textbf{in} \\
 & & \textbf{let}\ \ y_1 = x_1 \cdot x_1 \\
 & & \qquad y_2 = h\ x_2 \\
 & & \qquad z_1 = (y_1, y_2) \\
 & & \textbf{in}\ \ ::\, z_1 \\
 & & \textbf{end}
\end{array} \\
\qquad\qquad\quad \textbf{end})
\end{array}
$$