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:

$\rightarrow$ Die Ausführung zweier VLIWs kann überlappen !!!

$\rightarrow$ 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:

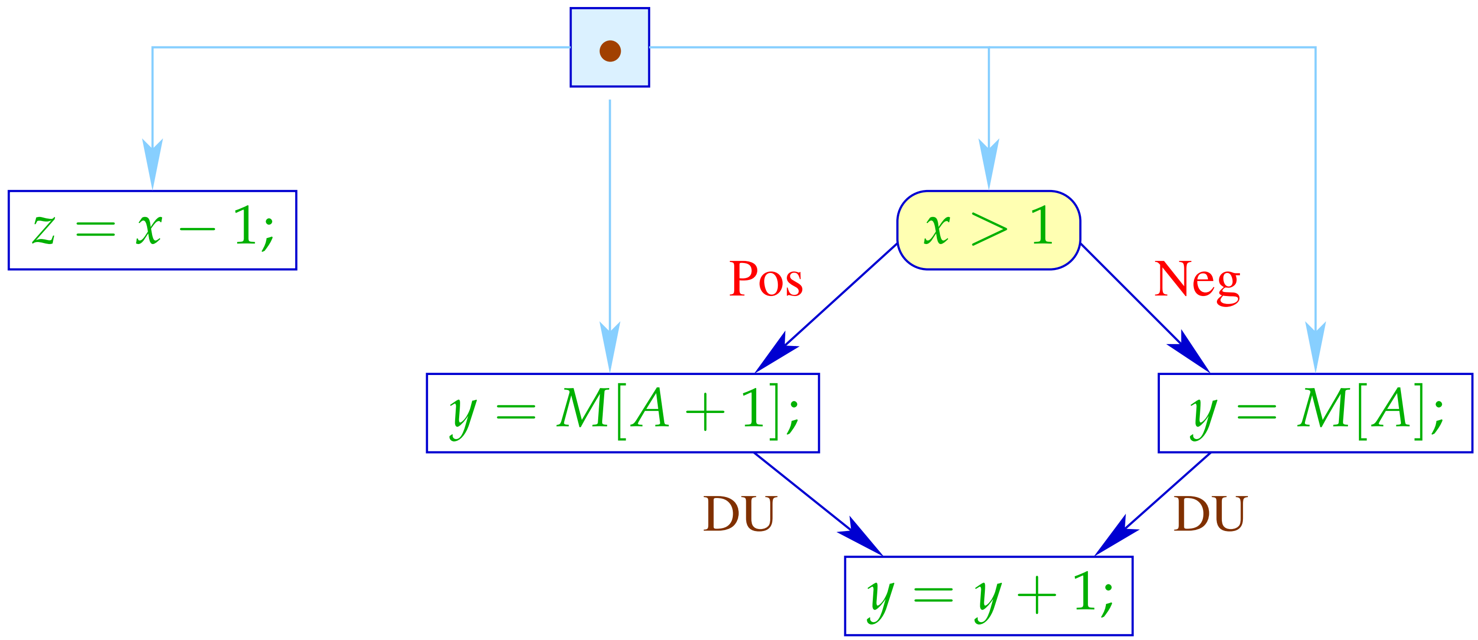
- Wenn Instruktionen zukünftiger Wortwahl weitere Restriktionen auferlegen, vermerken wir diese ebenfalls in Σ .
- Trotzdem unterscheiden wir nur endlich viele System-Zustände :-)
- Die Berechnung des Effekts eines VLIW auf Σ lässt sich in einen endlichen Automaten compilieren !!!
- Dieser Automat könnte 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

$$
\begin{array}{l}
\textsf{if}\ \ (x > 1)\ \ \{ \\
\qquad y = M[A]; \\
\qquad z = x - 1; \\
\}\ \ \textsf{else}\ \ \{ \\
\qquad y = M[A+1]; \\
\qquad z = x - 1; \\
\} \\
y = y + 1;
\end{array}
$$

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:

	$z = x - 1$	$\mathsf{if}\ \ (!(x > 0))\ \ \mathsf{goto}\ \ A$
	$y = M[A]$	
	$\mathsf{goto}\ \ B$	
$A:$	$y = M[A+1]$	
$B:$	$y = y + 1$	

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:

	$z = x - 1$	$y = M[A]$	$\textsf{if}\ \ (x > 0)\ \ \textsf{goto}\ \ B$
	$y = M[A+1]$		
$B:$			
	$y = y + 1$		

Im Fall $x \leq 0$ haben wir $y = M[A]$ zuviel ausgeführt.

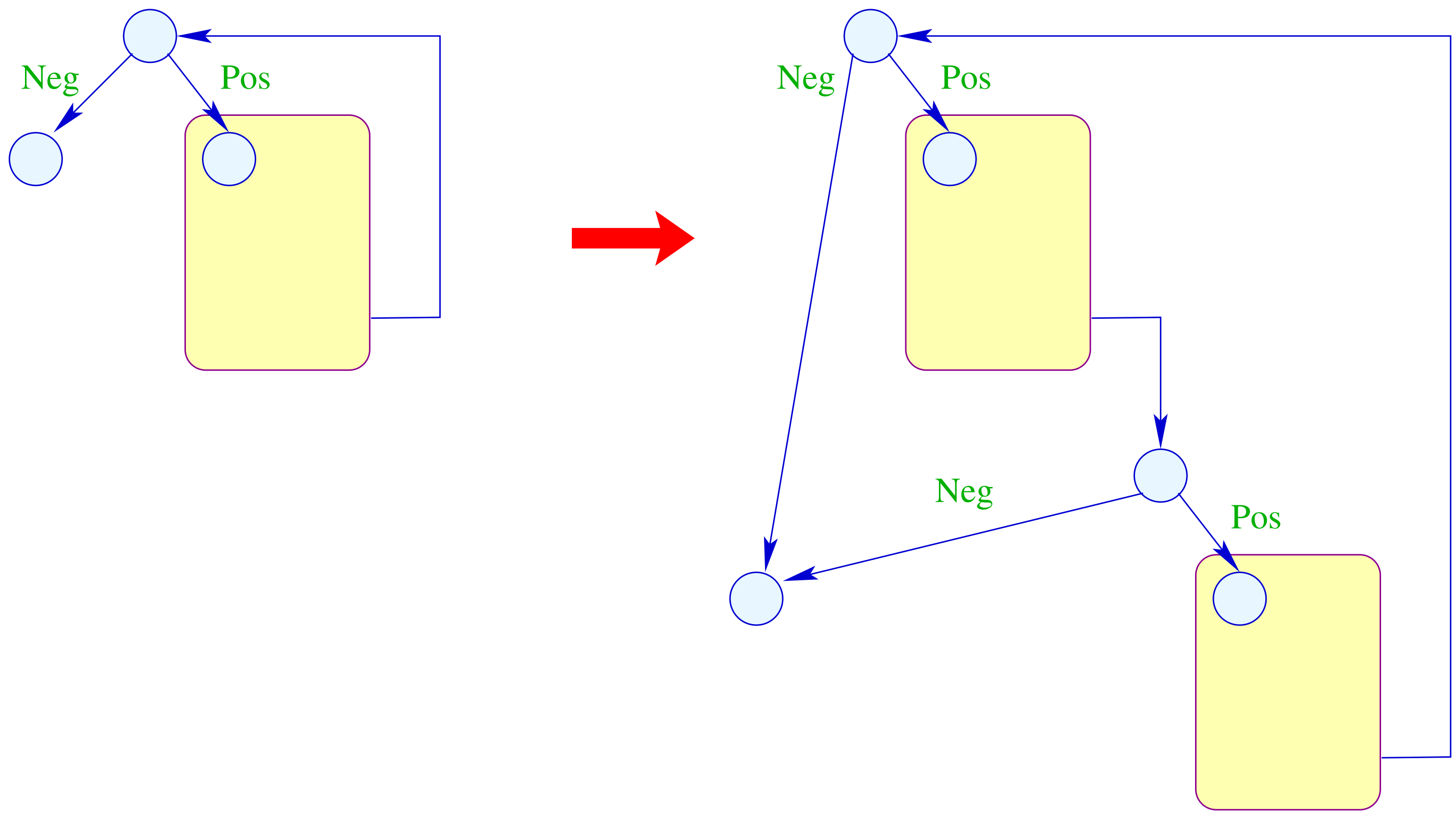
Dieser Wert wird aber im nächsten Schritt direkt überschrieben :-)

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:

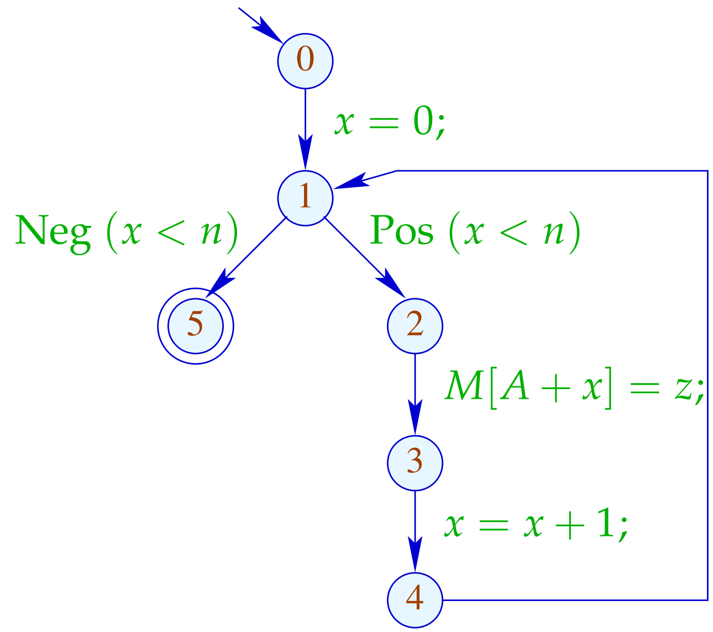
diejenige, die innerhalb des abgerollten Rumpfs der Schleife bleibt :-)

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!

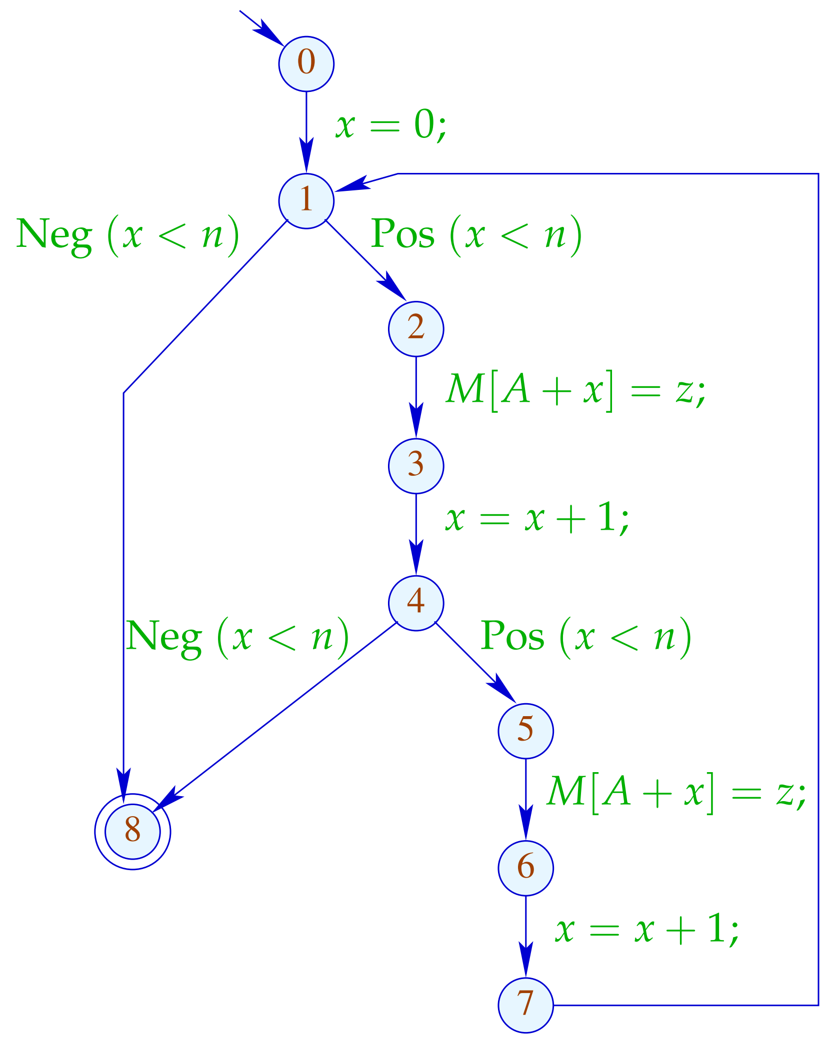
Beispiel:

$$\text{for } (x = 0; x < n; x{+}{+})$$
$$M[A + x] = z;$$



Verdoppelung des Rumpfs liefert:

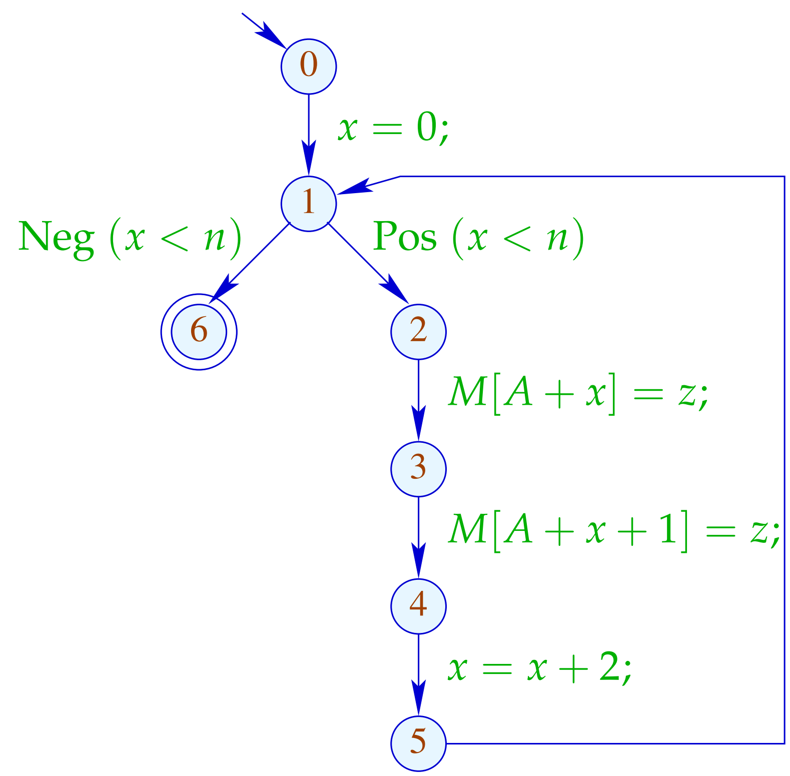
$$
\begin{array}{l}
\text{for } (x = 0; x < n; x{+}{+}) \ \{ \\
\qquad M[A+x] = z; \\
\qquad x = x + 1; \\
\qquad \text{if } (!(x < n)) \ \text{break}; \\
\qquad M[A+x] = z; \\
\}
\end{array}
$$



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:

$$
\begin{array}{l}
\textsf{for}\ \ (x = 0; x < n; x = x + 2)\ \ \{ \\
\qquad\qquad M[A + x] = z; \\
\qquad\qquad M[A + x + 1] = z; \\
\}
\end{array}
$$



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:

$$
\begin{array}{ll}
\textsf{for}\ (x = 0; x < n; x{+}{+})\ \{ & \qquad \textsf{for}\ (x = 0; x < n; x{+}{+})\ \{ \\
\qquad R = M[B + x]; & \qquad\qquad R = M[B + x]; \\
\qquad S = M[C + x]; & \qquad\qquad S = M[C + x]; \\
\qquad T_1 = R + S; & \qquad\qquad T_2 = R - S; \\
\qquad M[A + x] = T_1; & \qquad\qquad M[C + x] = T_2; \\
\} & \qquad \}
\end{array}
$$

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:

$$
\begin{array}{ll}
\multicolumn{2}{l}{\mathsf{for}\ \ (x = 0; x < n; x{+}{+})\ \ \{} \\
\qquad R = M[B+x]; & \qquad R = M[B+x]; \\
\qquad S = M[C+x]; & \qquad S = M[C+x]; \\
\qquad T_1 = R + S; & \qquad T_2 = R - S; \\
\qquad M[A+x] = T_1; & \qquad M[C+x] = T_2; \\
\multicolumn{2}{c}{\}}
\end{array}
$$

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:

$$\begin{array}{lll} x_{\mathsf{write}} & \geq & C \\ x_{\mathsf{write}} & \leq & C + x - 1 \\ x_{\mathsf{read}} & = & C + x \\ x_{\mathsf{read}} & = & x_{\mathsf{write}} \end{array}$$

... hat offenbar keine Lösung :-)

Allgemeine Form:

$$\begin{aligned} x &\geq t_1 \\ t_2 &\geq x \\ y &= s \\ x &= y \end{aligned}$$

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:

$$\begin{array}{lll} 0 & \leq & 13 + 7 \cdot x \\ 0 & \leq & -1 + 5 \cdot x \end{array}$$

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:

$$\begin{array}{lll} 0 & \leq & 13 - 7 \cdot x \\ 0 & \leq & -1 + 5 \cdot x \end{array} \qquad \Longleftrightarrow \qquad \begin{array}{lll} x & \leq & \frac{13}{7} \\ 0 & \leq & -1 + 5 \cdot x \end{array}$$

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

$$\text{ggT}\{a_1, \ldots, a_k\} \quad \Big| \quad b$$

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:

$$\begin{array}{rcrcrcr} 2x & + & 3y & & & = & 24 \\ {\color{blue}x} & - & y & + & 5z & = & 3 \end{array}$$

$$\Longrightarrow \qquad {\color{blue}x} = 3 + y - 5z$$

Beispiel:

$$2x + 3y = 24$$

$$x = 3 + y - 5z$$

Beispiel:

$$2x + 3y = 24$$

$$x = 3 + y - 5z$$

$$\Longrightarrow$$

$$5y - 10z = 18$$