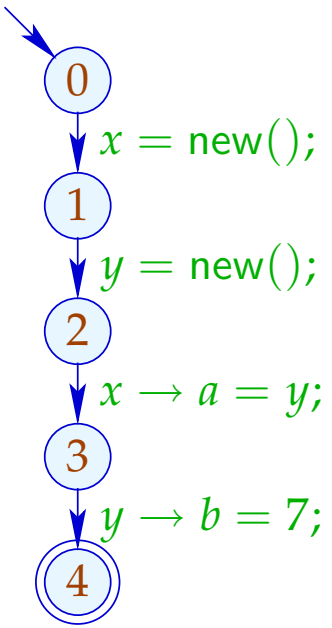
...

- 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:




| | |
|---|---|
| $x$ | $\{(0,1)\}$ |
| $y$ | $\{(1,2)\}$ |
| $(0,1).a$ | $\{(1,2)\}$ |
| $(0,1).b$ | $\emptyset$ |

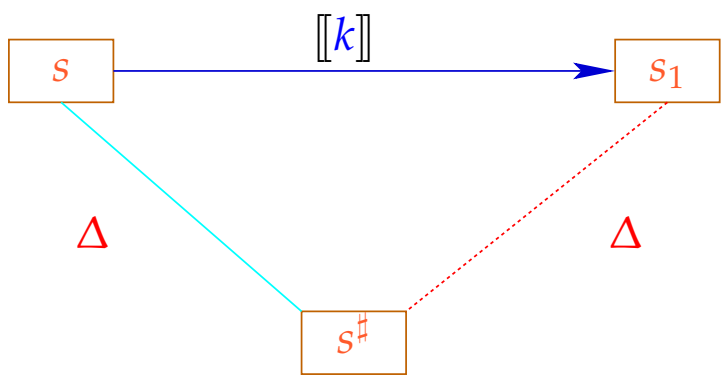
Jede Kante $(u, lab, v)$ gibt Anlass zu Constraints:

| *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 $f \in Addr^\sharp$ |

Andere Kanten haben keinen Effekt :-)

# Diskussion:

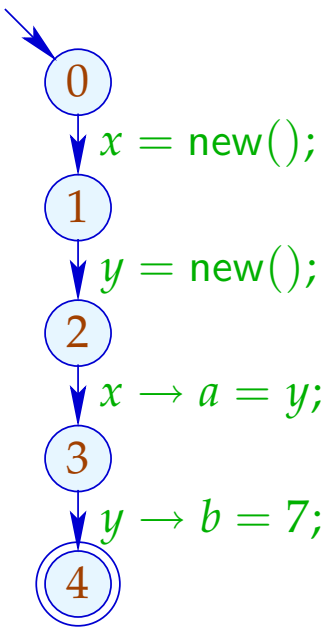
- Das resultierende Constraint-System ist $\mathcal{O}(k \cdot n)$ bei $k$ abstrakten Adressen und $n$ Kanten :-(
- Die Anzahl eventuell notwendiger Iterationen ist $\mathcal{O}(k)$ ...
- Die berechnete Information ist möglicherweise immer noch zu präzise !!?
- Zur Korrektheit einer Lösung $s^\sharp \in States^\sharp$ 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 $\pi$ sind:

$$
\begin{array}{lcll}
\mathsf{find}\,(\pi, p) & = & p_i & \text{falls } p \in P_i \\
 & // & \text{liefert den Repräsentanten} & \\
\mathsf{union}\,(\pi, p_{i_1}, p_{i_2}) & = & \{P_{i_1} \cup P_{i_2}\} \cup \{P_j \mid i_1 \neq j \neq i_2\} & \\
 & // & \text{vereinigt repräsentierte Klassen} &
\end{array}
$$

$\rightarrow$ Sind $x_1, x_2 \in \mathit{Vars}$ äquivalent, müssen auch $x_i \rightarrow a$ und $x_i \rightarrow b$ äquivalent sein :-)

$\rightarrow$ Ist $P_i \cap \mathit{Vars} \neq \emptyset$ , soll auch $p_i \in \mathit{Vars}$ gelten. Dann können wir union rekursiv anwenden :

$$
\begin{array}{lll}
\mathsf{union}^*\,(\pi, q_1, q_2) & = & \mathsf{let}\ \ p_{i_1} = \mathsf{find}\,(\pi, q_1) \\
& & \qquad\ \ p_{i_2} = \mathsf{find}\,(\pi, q_2) \\
& & \mathsf{in}\ \ \mathsf{if}\ p_{i_1} == p_{i_2}\ \mathsf{then}\ \pi \\
& & \qquad \mathsf{else}\ \ \mathsf{let}\ \pi = \mathsf{union}\,(\pi, p_{i_1}, p_{i_2}) \\
& & \qquad\qquad \mathsf{in}\ \ \mathsf{if}\ p_{i_1}, p_{i_2} \in \mathit{Vars}\ \mathsf{then} \\
& & \qquad\qquad\qquad \mathsf{let}\ \pi = \mathsf{union}^*\,(\pi, p_{i_1} \rightarrow a, p_{i_2} \rightarrow a) \\
& & \qquad\qquad\qquad \mathsf{in}\ \mathsf{union}^*\,(\pi, p_{i_1} \rightarrow b, p_{i_2} \rightarrow b) \\
& & \qquad\qquad \mathsf{else}\ \pi
\end{array}
$$

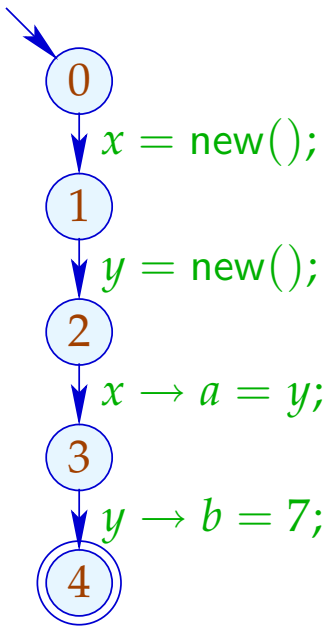
Die Analyse iteriert einmal über alle Kanten:

$$\pi = \{\{x\}, \{x \to a\}, \{x \to b\} \mid x \in \mathit{Vars}\};$$
$$\mathsf{forall}\, k = (\_, \mathit{lab}, \_)\, \mathsf{do} \quad \pi = [\![\mathit{lab}]\!]^\sharp\, \pi;$$

Dabei ist:

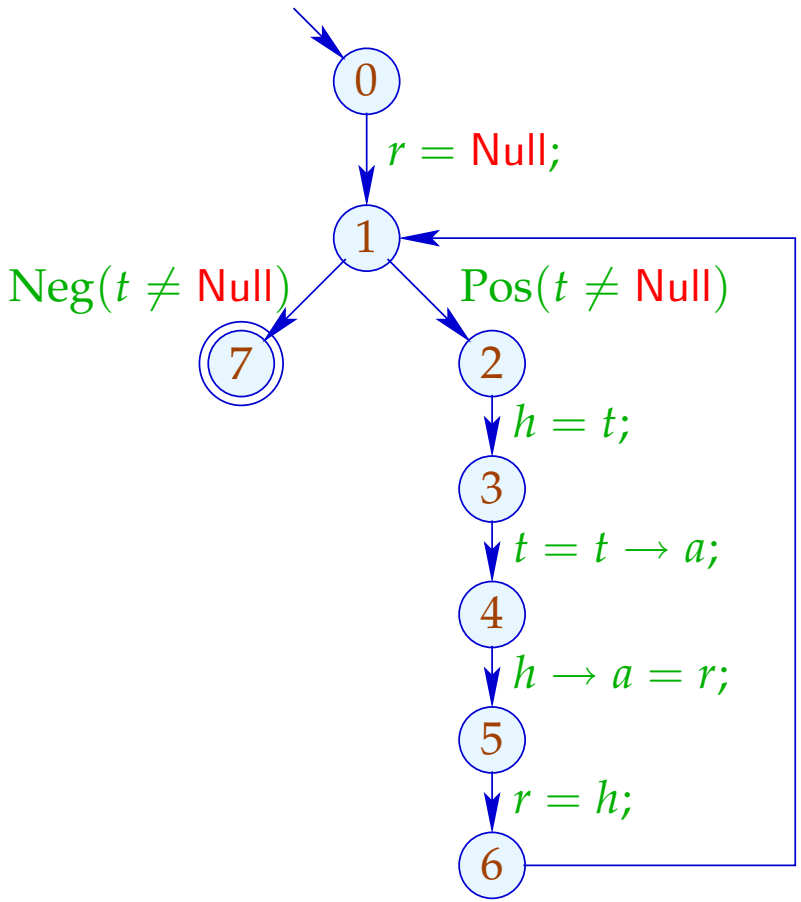
$$\begin{array}{lcll}
[\![x = y;]\!]^\sharp\, \pi & = & \mathsf{union}^*\,(\pi, x, y) & \\
[\![x = R \to a;]\!]^\sharp\, \pi & = & \mathsf{union}^*\,(\pi, x, R \to a) & \\
[\![R \to a = x;]\!]^\sharp\, \pi & = & \mathsf{union}^*\,(\pi, x, R \to a) & \\
[\![\mathit{lab}]\!]^\sharp\, \pi & = & \pi & \text{sonst}
\end{array}$$

# ... im einfachen Beispiel:




| | |
|---|---|
| | $\{\{x\}, \{y\}, \{x \to a\}, \{y \to a\}, \ldots\}$ |
| $(0,1)$ | $\{\{x\}, \{y\}, \{x \to a\}, \{y \to a\}, \ldots\}$ |
| $(1,2)$ | $\{\{x\}, \{y\}, \{x \to a\}, \{y \to a\}, \ldots\}$ |
| $(2,3)$ | $\{\{x\}, \boxed{\{y, x \to a\}}, \{y \to a\}, \ldots\}$ |
| $(3,4)$ | $\{\{x\}, \{y, x \to a\}, \{y \to a\}, \ldots\}$ |

# ... 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:

| | | |
|---|---|---|
| $\mathcal{O}(\#\,Kanten)$ | Aufrufe von | union$^*$ |
| $\mathcal{O}(\#\,Kanten)$ | Aufrufe von | find |
| $\mathcal{O}(\#\,Vars)$ | Aufrufe von | union |

$\Longrightarrow$ Wir benötigen effiziente Union-Find-Datenstruktur :-)

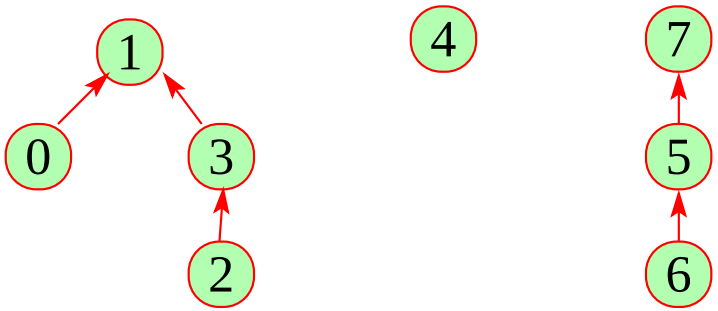
## Idee:

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 ...

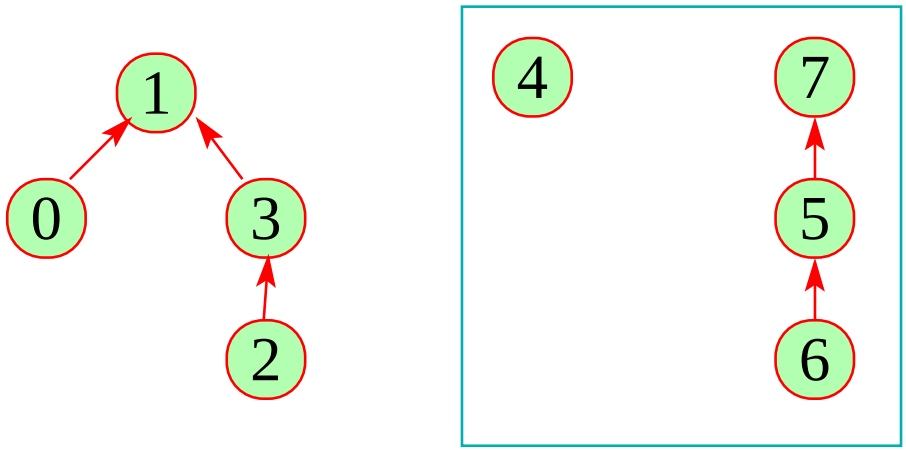



| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|

| 1 | 1 | 3 | 1 | 4 | 7 | 5 | 7 |
|---|---|---|---|---|---|---|---|

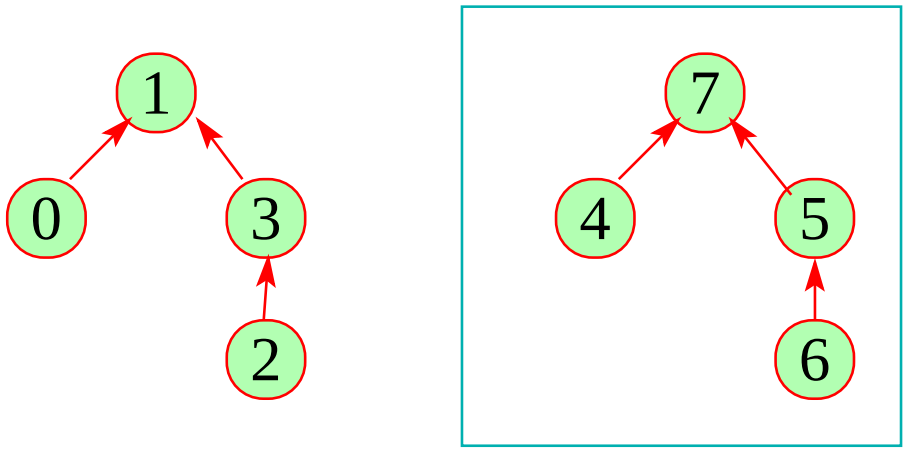
$\longrightarrow$ find $(\pi, u)$ folgt den Vater-Verweisen :-)

$\longrightarrow$ union $(\pi, u_1, u_2)$ hängt den Vater-Verweis eines $u_i$ um ...




| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|

| 1 | 1 | 3 | 1 | 4 | 7 | 5 | 7 |
|---|---|---|---|---|---|---|---|




| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|

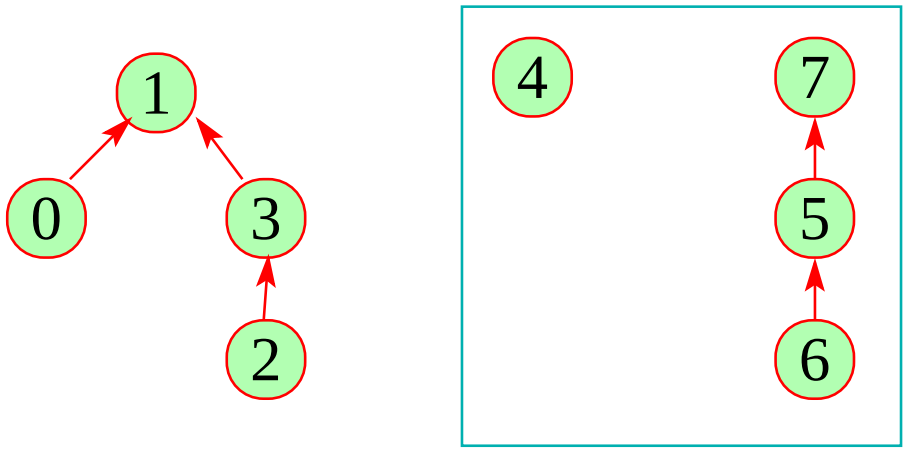
| 1 | 1 | 3 | 1 | 7 | 7 | 5 | 7 |
|---|---|---|---|---|---|---|---|

## Die Kosten:

| | | | |
|---|---|---|---|
| union | : | $\mathcal{O}(1)$ | :-) |
| find | : | $\mathcal{O}(depth(\pi))$ | :-( |

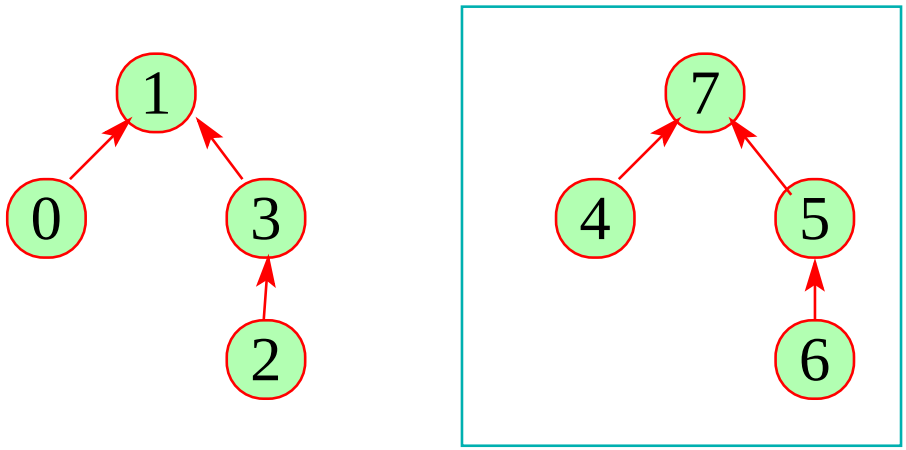
## Strategie zur Vermeidung tiefer Bäume:

- Hänge den kleineren Baum unter den größeren !
- Benutze find , um Pfade zu komprimieren ...



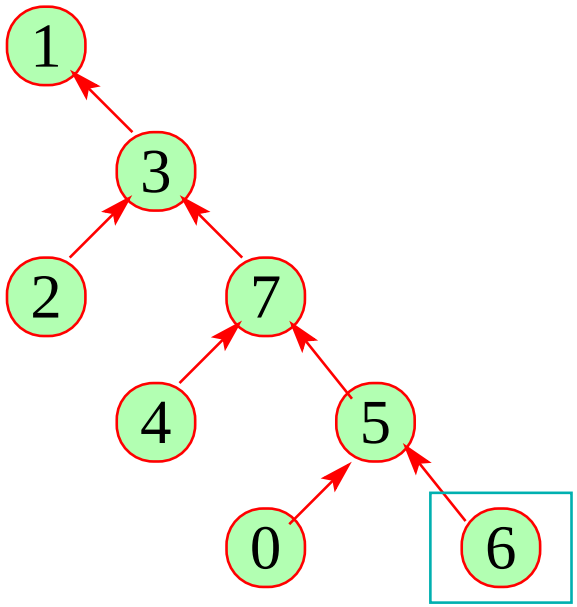

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|

| 1 | 1 | 3 | 1 | 4 | 7 | 5 | 7 |
|---|---|---|---|---|---|---|---|

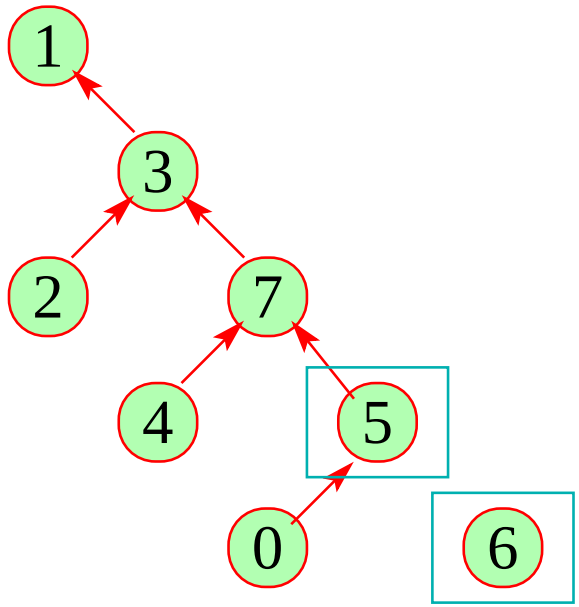



| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|

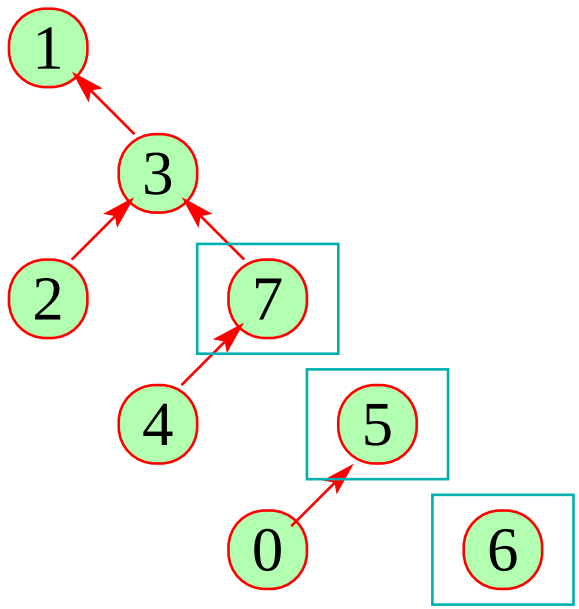
| 1 | 1 | 3 | 1 | 7 | 7 | 5 | 7 |
|---|---|---|---|---|---|---|---|



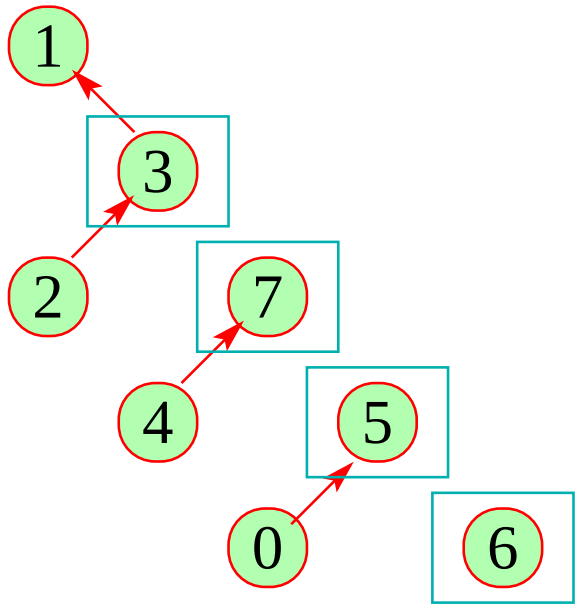

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 5 | 1 | 3 | 1 | 7 | 7 | 5 | 3 |



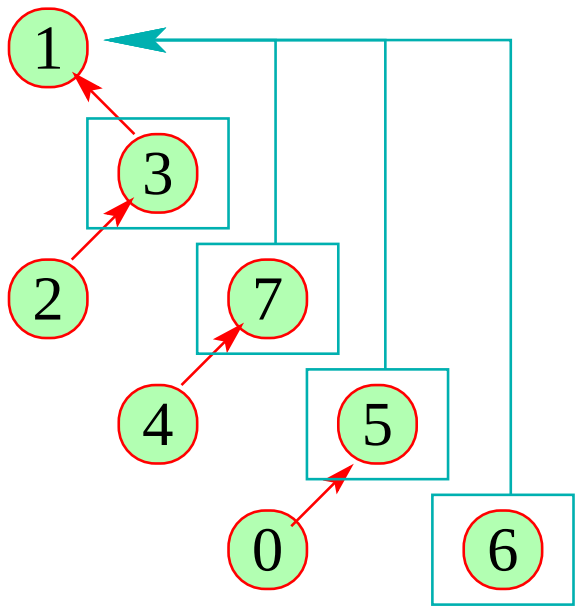

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 5 | 1 | 3 | 1 | 7 | 7 | 5 | 3 |



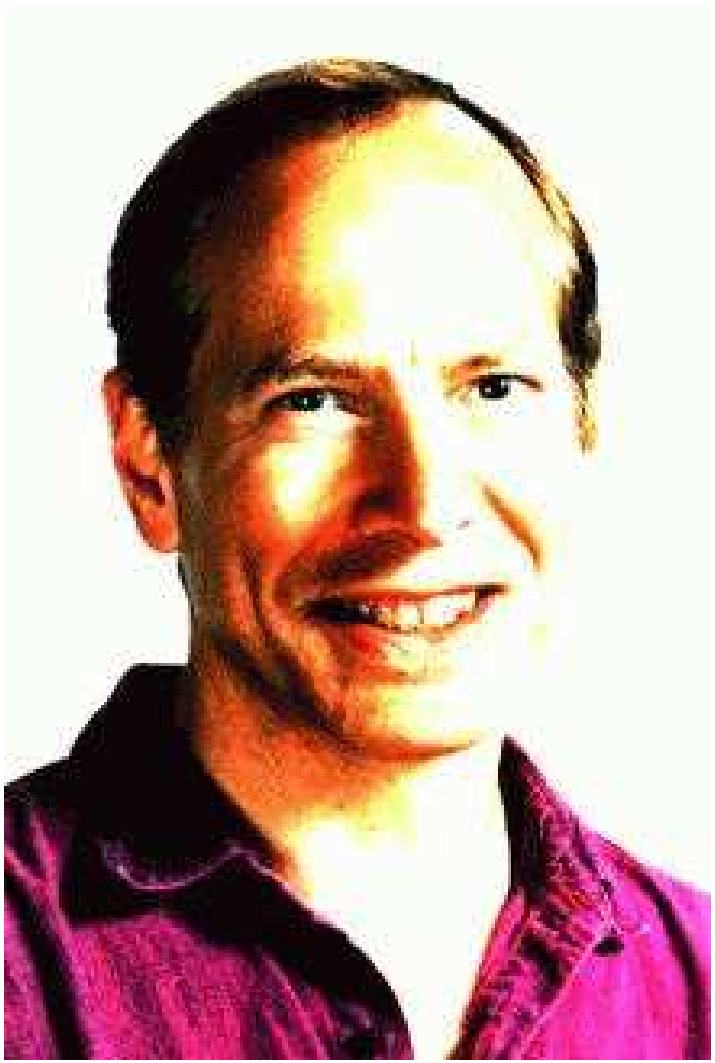

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 5 | 1 | 3 | 1 | 7 | 7 | 5 | 3 |




| 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 $\mathcal{O}(n + m \cdot \alpha(n, n))$

  // $\alpha$ 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:

- $\rightarrow$ Wir benötigen eine ganze Runde, um Terminierung festzustellen :-(
- $\rightarrow$ Ändert sich in einer Runde der Wert nur einer Variable, berechnen wir trotzdem alle neu :-(
- $\rightarrow$ 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:

- $\rightarrow$ 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.
- $\rightarrow$ die Werte $D[x_i]$ der $x_i$, wobei anfangs $D[x_i] = \bot$;
- $\rightarrow$ Eine Menge $W$ der Variablen, deren Wert neu berechnet werden muss ...

# Der Algorithmus:

$W = \{x_1, \ldots, x_n\};$

while $(W \neq \emptyset)$ {

$x_i$ = extract $W$;

$t$ = $f_i$ eval;

if $(t \not\sqsubseteq D[x_i])$ {

$D[x_i] = D[x_i] \sqcup t;$

$W = W \cup I[x_i];$

}

}

wobei :

$eval\ x_j = D[x_j]$

## Beispiel:

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

| | $I$ |
|---|---|
| $x_1$ | $\{x_3\}$ |
| $x_2$ | $\emptyset$ |
| $x_3$ | $\{x_1, x_2\}$ |

# Beispiel:

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

| | $I$ |
|---|---|
| $x_1$ | $\{x_3\}$ |
| $x_2$ | $\emptyset$ |
| $x_3$ | $\{x_1, x_2\}$ |

| $D[x_1]$ | $D[x_2]$ | $D[x_3]$ | $W$ |
|---|---|---|---|
| $\emptyset$ | $\emptyset$ | $\emptyset$ | $\boxed{x_1}, x_2, x_3$ |
| $\{a\}$ | $\emptyset$ | $\emptyset$ | $\boxed{x_2}, x_3$ |
| $\{a\}$ | $\emptyset$ | $\emptyset$ | $\boxed{x_3}$ |
| $\{a\}$ | $\emptyset$ | $\{a, c\}$ | $\boxed{x_1}, x_2$ |
| $\{a, c\}$ | $\emptyset$ | $\{a, c\}$ | $\boxed{x_3}, x_2$ |
| $\{a, c\}$ | $\emptyset$ | $\{a, c\}$ | $\boxed{x_2}$ |
| $\{a, c\}$ | $\{a\}$ | $\{a, c\}$ | $\emptyset$ |

## 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

$$N = \sum_{i=1}^{n} (1 + \#\,(\mathit{Dep}\; f_i)) \qquad // \quad \text{Größe des Systems} \quad \text{:-)}$$

(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:

$$\boxed{D[x_i] = D[x_i] \sqcup t;} \quad \Longrightarrow \quad \boxed{D[x_i] = \qquad\qquad t;}$$

- 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:

$$\boxed{D[x_i] = D[x_i] \sqcup t;} \quad \Longrightarrow \quad \boxed{D[x_i] = D[x_i] \sqcap t;}$$

## 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 :-)

$$\begin{aligned} \mathsf{eval}\ x_i\ x_j \quad = \quad & \mathsf{solve}\, x_j; \\ & I[x_j] = I[x_j] \cup \{x_i\}; \\ & D[x_j]; \end{aligned}$$

→ 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 :

$$
\begin{array}{lll}
\text{solve } x_i & = & \text{if } (x_i \notin \mathit{Stable}) \; \{ \\
& & \quad \mathit{Stable} = \mathit{Stable} \cup \{x_i\}; \\
& & \quad t = f_i \; (\text{eval } \; x_i); \\
& & \quad \text{if } (t \not\sqsubseteq D[x_i]) \; \{ \\
& & \qquad W = I[x_i]; \quad I[x_i] = \emptyset; \\
& & \qquad D[x_i] = D[x_i] \sqcup t; \\
& & \qquad \mathit{Stable} = \mathit{Stable} \backslash W; \\
& & \qquad \text{app } \; \text{solve } \; W; \\
& & \quad \} \\
& & \} \\
\end{array}
$$

Helmut Seidl, TU München ;-)

## Beispiel:

Betrachte unser Standard-Beispiel:

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

Dann sieht ein Trace des Fixpunkt-Algorithmus etwa so aus: