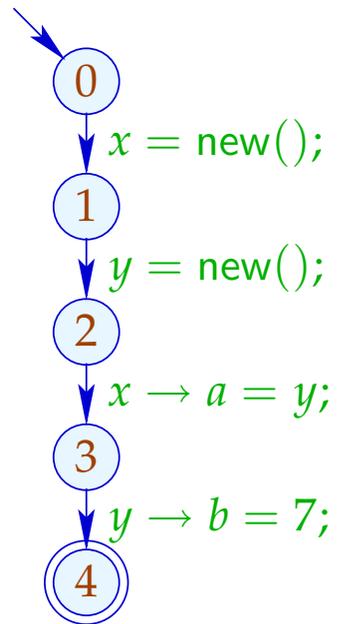


Alias-Analyse

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

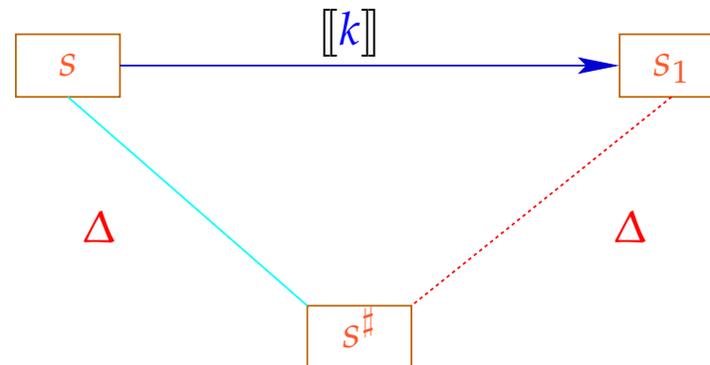
Jede Kante (u, lab, v) gibt Anlass zu Constraints:

<i>lab</i>	<i>Constraints</i>
$x = y;$	$\mathcal{P}[x] \supseteq \mathcal{P}[y]$
$x = \text{new}();$	$\mathcal{P}[x] \supseteq \{(u, v)\}$
$x = R \rightarrow a;$	$\mathcal{P}[x] \supseteq \bigcup \{\mathcal{P}[f.a] \mid f \in \mathcal{P}[R]\}$
$R \rightarrow a = x;$	$\mathcal{P}[f.a] \supseteq (f \in \mathcal{P}[R]) ? \mathcal{P}[x] : \emptyset$ für alle $f \in \text{Addr}^\#$

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 $s^\# \in States^\#$ des Constraint-Systems zeigt man:

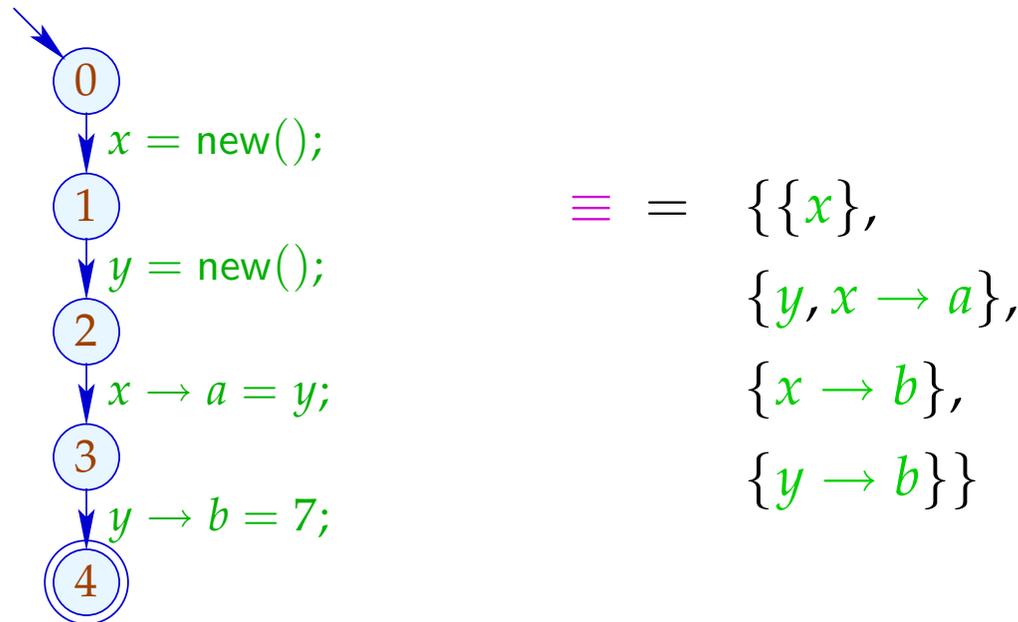


Alias-Analyse

3. Idee:

Berechne eine Äquivalenzrelation \equiv auf Variablen x und Selektoren $y \rightarrow a$ mit $s_1 \equiv s_2$ falls an irgendeinem u s_1, s_2 die gleiche Adresse enthalten ...

... im einfachen Beispiel:



Diskussion:

- Wir berechnen **eine Information** für das ganze Programm.
- Die Berechnung dieser Information verwaltet **Partitionen**
 $\pi = \{P_1, \dots, P_m\}$:-)
- Einzelne Mengen P_i identifizieren wir durch einen **Repräsentanten** $p_i \in P_i$.
- Die Operationen auf einer Partition π sind:

$$\text{find}(\pi, p) = p_i \quad \text{falls } p \in P_i$$

// liefert den Repräsentanten

$$\text{union}(\pi, p_{i_1}, p_{i_2}) = \{P_{i_1} \cup P_{i_2}\} \cup \{P_j \mid i_1 \neq j \neq i_2\}$$

// vereinigt repräsentierte Klassen

- Sind $x_1, x_2 \in Vars$ äquivalent, müssen auch $x_i \rightarrow a$ und $x_i \rightarrow b$ äquivalent sein :-)
- Ist $P_i \cap Vars \neq \emptyset$, soll auch $p_i \in Vars$ gelten. Dann können wir **union** **rekursiv** anwenden :

```

union* ( $\pi, q_1, q_2$ ) = let  $p_{i_1} = \text{find}(\pi, q_1)$ 
                           $p_{i_2} = \text{find}(\pi, q_2)$ 
in if  $p_{i_1} == p_{i_2}$  then  $\pi$ 
   else let  $\pi = \text{union}(\pi, p_{i_1}, p_{i_2})$ 
        in if  $p_{i_1}, p_{i_2} \in Vars$  then
            let  $\pi = \text{union}^*(\pi, p_{i_1} \rightarrow a, p_{i_2} \rightarrow a)$ 
            in union* ( $\pi, p_{i_1} \rightarrow b, p_{i_2} \rightarrow b$ )
        else  $\pi$ 

```

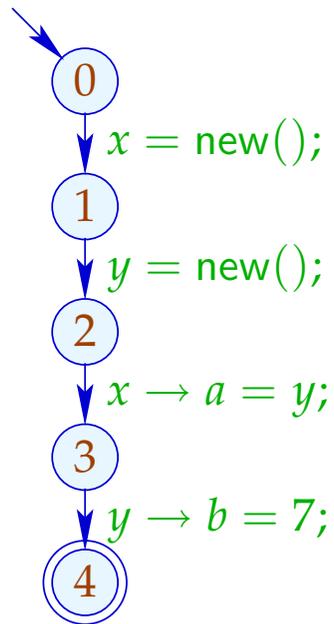
Die Analyse iteriert **einmal** über alle Kanten:

$$\begin{aligned} \pi &= \{ \{x\}, \{x \rightarrow a\}, \{x \rightarrow b\} \mid x \in \text{Vars} \}; \\ \text{forall } k &= (_, \text{lab}, _) \text{ do } \pi = \llbracket \text{lab} \rrbracket^\# \pi; \end{aligned}$$

Dabei ist:

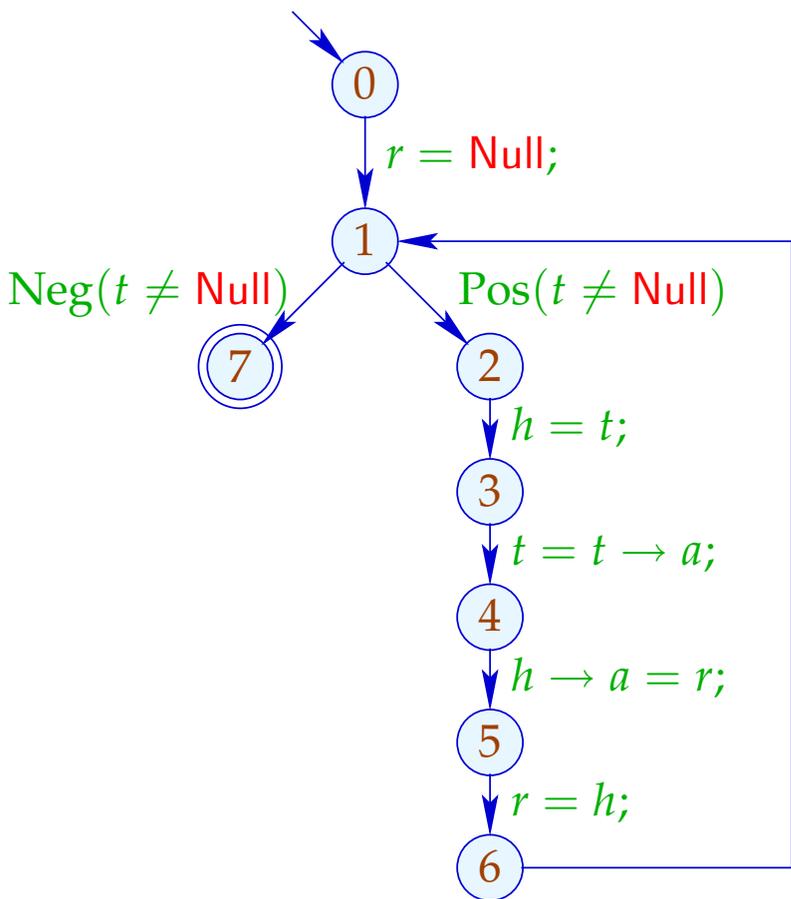
$$\begin{aligned} \llbracket x = y; \rrbracket^\# \pi &= \text{union}^* (\pi, x, y) \\ \llbracket x = R \rightarrow a; \rrbracket^\# \pi &= \text{union}^* (\pi, x, R \rightarrow a) \\ \llbracket R \rightarrow a = x; \rrbracket^\# \pi &= \text{union}^* (\pi, x, R \rightarrow a) \\ \llbracket \text{lab} \rrbracket^\# \pi &= \pi \quad \text{sonst} \end{aligned}$$

... im einfachen Beispiel:



	$\{\{x\}, \{y\}, \{x \rightarrow a\}, \{y \rightarrow a\}, \dots\}$
$(0, 1)$	$\{\{x\}, \{y\}, \{x \rightarrow a\}, \{y \rightarrow a\}, \dots\}$
$(1, 2)$	$\{\{x\}, \{y\}, \{x \rightarrow a\}, \{y \rightarrow a\}, \dots\}$
$(2, 3)$	$\{\{x\}, \{y, x \rightarrow a\}, \{y \rightarrow a\}, \dots\}$
$(3, 4)$	$\{\{x\}, \{y, x \rightarrow a\}, \{y \rightarrow a\}, \dots\}$

... im komplizierten Beispiel:



	$\{\{h\}, \{r\}, \{t\}, \{h \rightarrow a\}, \{t \rightarrow a\}\}$
(2, 3)	$\{\{h, t\}, \{r\}, \{h \rightarrow a, t \rightarrow a\}\}$
(3, 4)	$\{\{h, t, h \rightarrow a, t \rightarrow a\}, \{r\}\}$
(4, 5)	$\{\{h, t, r, h \rightarrow a, t \rightarrow a\}\}$
(5, 6)	$\{\{h, t, r, h \rightarrow a, t \rightarrow a\}\}$

Achtung:

Um überhaupt etwas heraus zu kriegen, müssen wir annehmen, dass alle Variablen anfangs auf **verschiedene** Adressen zeigen.

Zur Komplexität:

Wir haben:

$O(\# \text{ Kanten})$	Aufrufe von	union*
$O(\# \text{ Kanten})$	Aufrufe von	find
$O(\# \text{ Vars})$	Aufrufe von	union

⇒ Wir benötigen effiziente **Union-Find-Datenstruktur** :-)

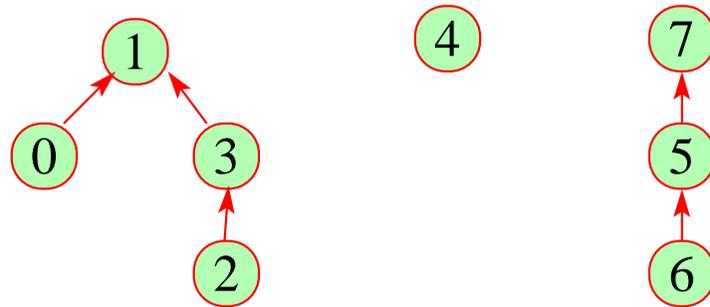
Idee:

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.

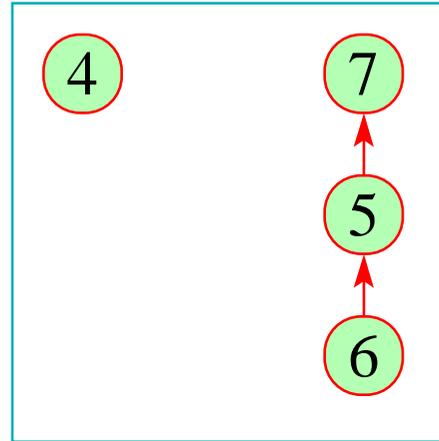
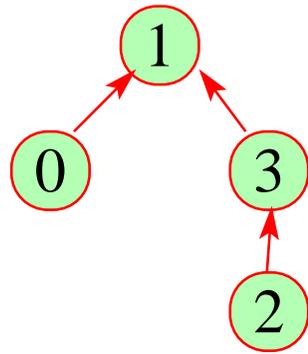
Ihre Wurzeln sind die Repräsentanten ...



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

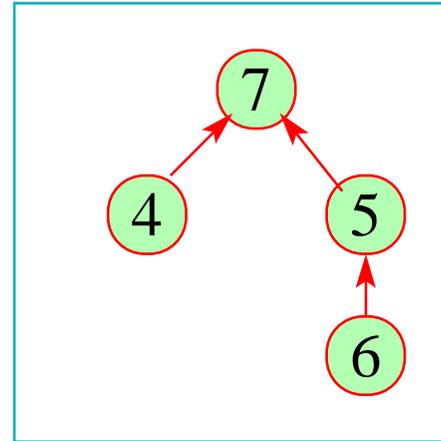
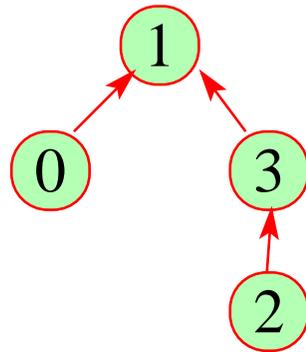
1	1	3	1	4	7	5	7
---	---	---	---	---	---	---	---

- $\text{find}(\pi, u)$ folgt den Vater-Verweisen :-)
- $\text{union}(\pi, u_1, u_2)$ 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
---	---	---	---	---	---	---	---

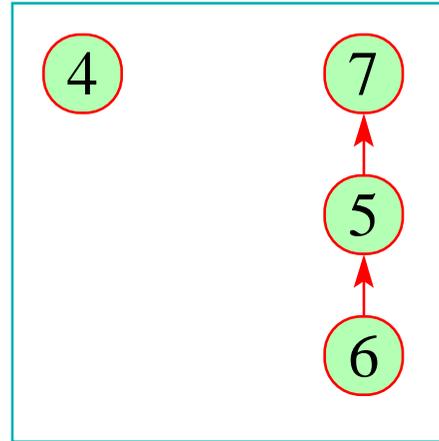
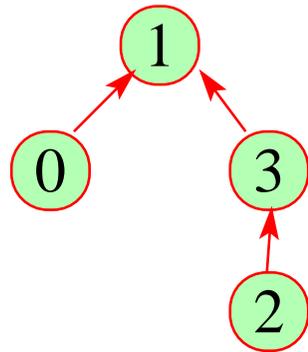
Die Kosten:

`union` : $\mathcal{O}(1)$:-)

`find` : $\mathcal{O}(\text{depth}(\pi))$:-)

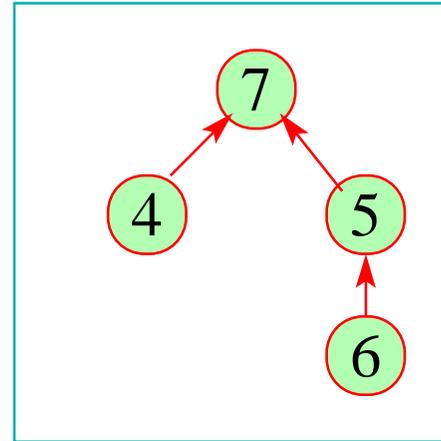
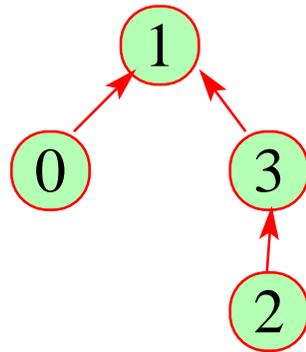
Strategie zur Vermeidung tiefer Bäume:

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



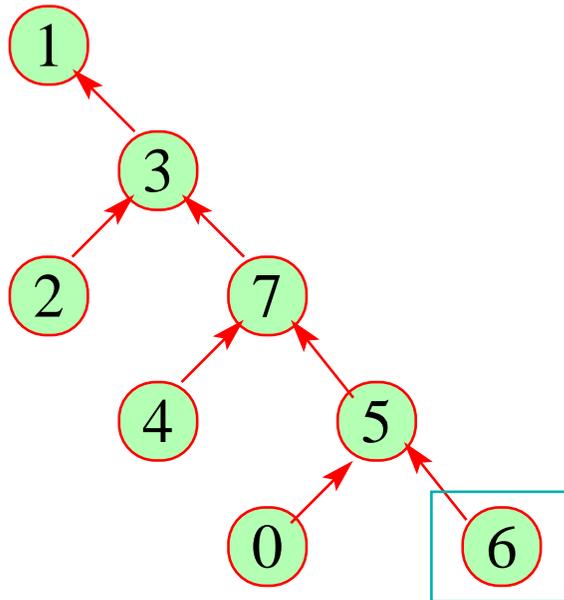
0	1	2	3	4	5	6	7
---	---	---	---	---	---	---	---

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

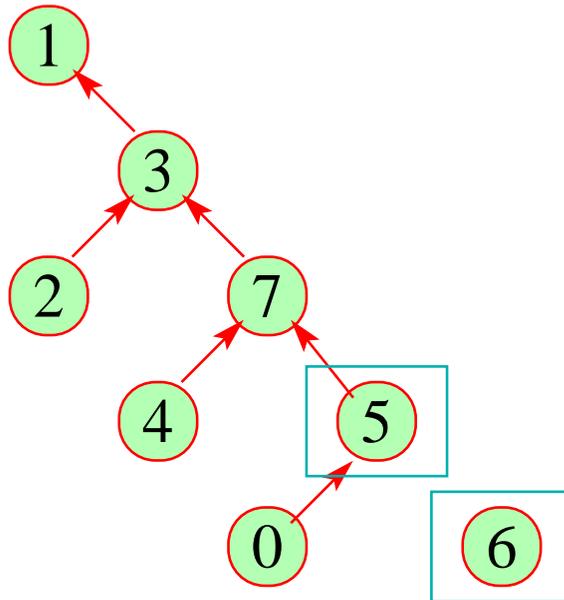


0	1	2	3	4	5	6	7
---	---	---	---	---	---	---	---

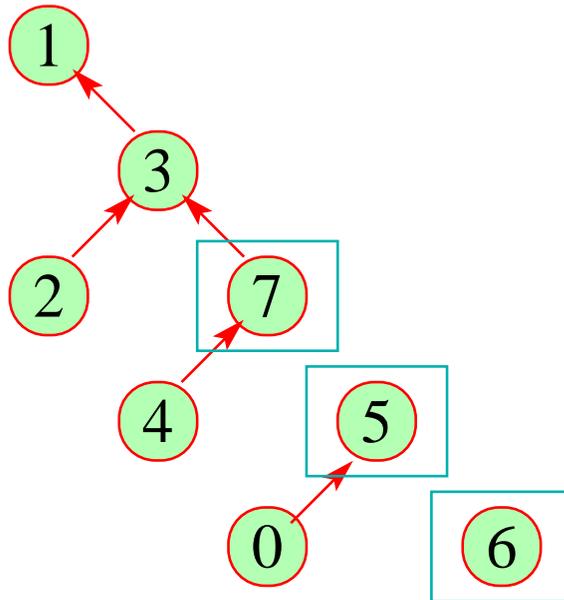
1	1	3	1	7	7	5	7
---	---	---	---	---	---	---	---



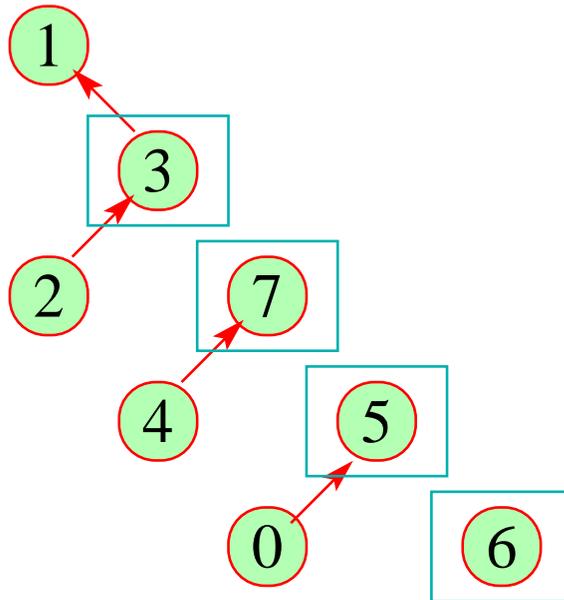
0	1	2	3	4	5	6	7
5	1	3	1	7	7	5	3



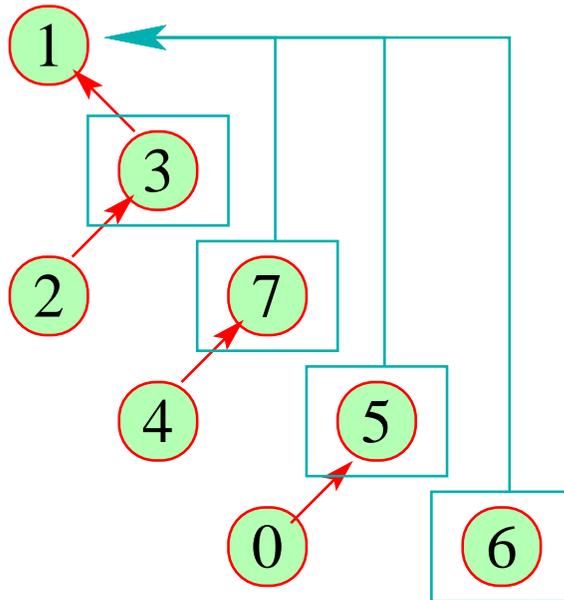
0	1	2	3	4	5	6	7
5	1	3	1	7	7	5	3



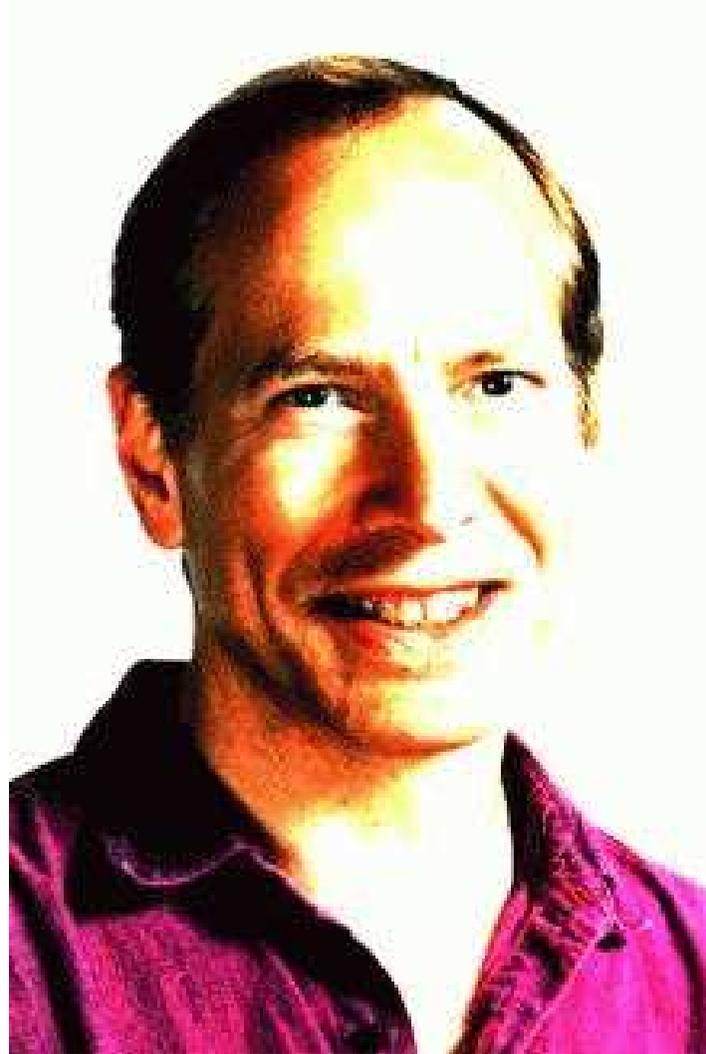
0	1	2	3	4	5	6	7
5	1	3	1	7	7	5	3



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 $\mathcal{O}(n + m \cdot \alpha(n, n))$
// 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: $x_i \supseteq f_i(x_1, \dots, x_n), \quad i = 1, \dots, n$

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:

```
 $W = \{x_1, \dots, x_n\};$   
while ( $W \neq \emptyset$ ) {  
     $x_i = \text{extract } W;$   
     $t = f_i \text{ eval};$   
    if ( $t \not\subseteq D[x_i]$ ) {  
         $D[x_i] = D[x_i] \sqcup t;$   
         $W = W \cup I[x_i];$   
    }  
}
```

wobei :

```
 $\text{eval } x_j = D[x_j]$ 
```

Beispiel:

$$x_1 \supseteq \{a\} \cup x_3$$

$$x_2 \supseteq x_3 \cap \{a, b\}$$

$$x_3 \supseteq x_1 \cup \{c\}$$

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

Beispiel:

$$x_1 \supseteq \{a\} \cup x_3$$

$$x_2 \supseteq x_3 \cap \{a, b\}$$

$$x_3 \supseteq x_1 \cup \{c\}$$

	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	x_1, x_2, x_3
$\{a\}$	\emptyset	\emptyset	x_2, x_3
$\{a\}$	\emptyset	\emptyset	x_3
$\{a\}$	\emptyset	$\{a, c\}$	x_1, x_2
$\{a, c\}$	\emptyset	$\{a, c\}$	x_3, x_2
$\{a, c\}$	\emptyset	$\{a, c\}$	x_2
$\{a, c\}$	$\{a\}$	$\{a, c\}$	\emptyset

Theorem

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

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

- (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:

- $D_0[x_i] \supseteq D[x_i]$ (zu jedem Zeitpunkt)
- $D[x_i] \not\supseteq f_i \text{ eval} \implies x_i \in W$ (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:

$$\boxed{D[x_i] = D[x_i] \sqcup t;} \implies \boxed{D[x_i] = t;}$$

- Für **Widening** ersetzt man:

$$\boxed{D[x_i] = D[x_i] \sqcup t;} \implies \boxed{D[x_i] = D[x_i] \sqcup\!\!\!\sqcup t;}$$

- Für **Narrowing** ersetzt man:

$$\boxed{D[x_i] = D[x_i] \sqcup t;} \implies \boxed{D[x_i] = D[x_i] \sqcap\!\!\!\sqcap t;}$$

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 ... :-)

\implies rekursive Auswertung ...