Bemerkung:

- Die möglichen Typ-Zuordnungen an Variablen und Programm-Ausdrücke erhalten wir als Lösung eines Gleichungssystems über Typ-Termen :-)
- Das Lösen von Systemen von Term-Gleichungen nennt man auch Unifikation :-)

Beispiel:

$$g(z, f(x)) = g(f(x), f(a))$$

Eine Lösung dieser Gleichung ist die Substitution $\{x \mapsto a, z \mapsto f(a)\}$

In dem Fall ist das offenbar die einzige :-)

Satz:

Jedes System von Term-Gleichungen:

$$s_i = t_i \qquad i = 1, \ldots, m$$

hat entweder keine Lösung oder eine allgemeinste Lösung.

Satz:

Jedes System von Term-Gleichungen:

$$s_i = t_i \qquad i = 1, \ldots, m$$

hat entweder keine Lösung oder eine allgemeinste Lösung.

Eine allgemeinste Lösung ist eine Substitution σ mit den Eigenschaften:

- σ ist eine Lösung, d.h. $\sigma(s_i) = \sigma(t_i)$ für alle i.
- σ ist allgemeinst, d.h. für jede andere Lösung τ gilt: $\tau = \tau' \circ \sigma$ für eine Substitution τ' :-)

Beispiele:

(1) $f(a) = g(x)$ — hat keine Lösung :-)

(2) $x = f(x)$ — hat ebenfalls keine Lösung ;-)

(3) $f(x) = f(a)$ — hat genau eine Lösung:-)

(4) $f(x) = f(g(y))$ — hat unendlich viele Lösungen :-)

(5) $x_0 = f(x_1, x_1), \ldots, x_{n-1} = f(x_n, x_n)$ —

hat mindestens exponentiell große Lösungen !!!

Bemerkungen:

- Es gibt genau eine Lösung, falls die allgemeinste Lösung keine Variablen enthält, d.h. ground ist :-)
- Gibt es zwei verschiedene Lösungen, dann bereits unendlich viele ;-)
- Achtung: Es kann mehrere allgemeinste Lösungen geben !!!

  Beispiel: $x = y$

  Allgemeinste Lösungen sind : $\{x \mapsto y\}$ oder $\{y \mapsto x\}$

  Diese sind allerdings nicht sehr verschieden :-)
- Eine allgemeinste Lösung kann immer idempotent gewählt werden, d.h. $\sigma = \sigma \circ \sigma$.

  Beispiel: $x = x \qquad y = y$

  Nicht idempotente Lösung: $\{x \mapsto y, y \mapsto x\}$

  Idempotente Lösung: $\{x \mapsto x, y \mapsto y\}$

Berechnung einer allgemeinsten Lösung:

$$
\begin{array}{lll}
\textbf{fun}\ \ \mathsf{occurs}\ (x, t) & = & \textbf{case}\ \ t \\
& & \quad \textbf{of}\ \ x \quad \longrightarrow \quad \mathsf{true} \\
& & \quad \mid\ \ f(t_1, \ldots, t_k) \quad \longrightarrow \quad \mathsf{occurs}\ (x, t_1) \vee \ldots \vee \mathsf{occurs}\ (x, t_k) \\
& & \quad \mid\ \ \_ \quad \longrightarrow \quad \mathsf{false} \\
\textbf{fun}\ \ \mathsf{unify}\ (s, t)\ \theta & = & \textbf{if}\ \ \theta\, s \equiv \theta\, t\ \ \textbf{then}\ \ \theta \\
& & \textbf{else}\ \ \textbf{case}\ \ (\theta\, s, \theta\, t) \\
& & \quad \textbf{of}\ \ (x, x) \quad \longrightarrow \quad \theta \\
& & \qquad (x, t) \quad \longrightarrow \quad \textbf{if}\ \ \mathsf{occurs}\ (x, t)\ \ \textbf{then}\ \ \mathsf{Fail} \\
& & \qquad\qquad\qquad\qquad \textbf{else}\ \ \{x \mapsto t\} \circ \theta \\
& & \quad \mid\ \ (t, x) \quad \longrightarrow \quad \textbf{if}\ \ \mathsf{occurs}\ (x, t)\ \ \textbf{then}\ \ \mathsf{Fail} \\
& & \qquad\qquad\qquad\qquad \textbf{else}\ \ \{x \mapsto t\} \circ \theta \\
& & \quad \mid\ \ (f(s_1, \ldots, s_k), f(t_1, \ldots, t_k)) \quad \longrightarrow \quad \mathsf{unifyList}\ [(s_1, t_1), \ldots, (s_k, t_k)]\ \theta \\
& & \quad \mid\ \ \_ \quad \longrightarrow \quad \mathsf{Fail}
\end{array}
$$

$$
\begin{array}{lll}
 & \ldots & \\
\textbf{and}\ \ \mathsf{unifyList}\ \mathsf{list}\ \theta & = & \textbf{case}\ \ \mathsf{list} \\
 & \textbf{of} & [\,] \ \longrightarrow\ \theta \\
 & \mid & ((s,t) :: \mathsf{rest}) \ \longrightarrow\ \ \textbf{let val}\ \ \theta = \mathsf{unify}\ (s,t)\ \theta \\
 & & \qquad\qquad\qquad\qquad\quad \textbf{in}\ \ \textbf{if}\ \ \theta = \mathsf{Fail}\ \ \textbf{then}\ \ \mathsf{Fail} \\
 & & \qquad\qquad\qquad\qquad\qquad\quad \textbf{else}\ \ \mathsf{unifyList}\ \mathsf{rest}\ \theta \\
 & & \qquad\qquad\qquad\qquad\quad \textbf{end}
\end{array}
$$

```
...
and unifyList list θ  =  case list
      of  [ ]  →  θ
       |  ((s,t) :: rest)  →  let val θ = unify (s,t) θ
                              in if θ = Fail then Fail
                                   else unifyList rest θ
                              end
```

Diskussion:

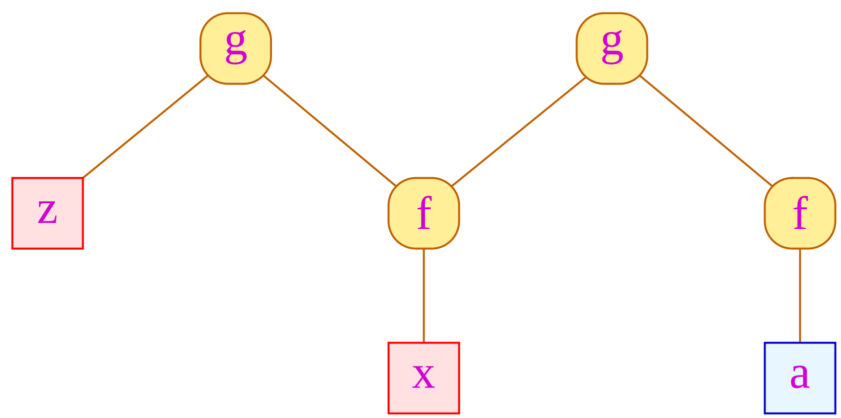
- Der Algorithmus startet mit unifyList $[(s_1, t_1), \ldots, (s_m, t_m)]$ $\{ \}$ ...
- Der Algorithmus liefert sogar eine idempotente allgemeinste Lösung :-)
- Leider hat er möglicherweise exponentielle Laufzeit :-(
- Lässt sich das verbessern ???

Idee:

- Wir repräsentieren die Terme der Gleichungen als Graphen.
- Dabei identifizieren wir bereits isomorphe Teilterme ;-)
- ...

... im Beispiel:

$$g(z, f(x)) = g(f(x), f(a))$$

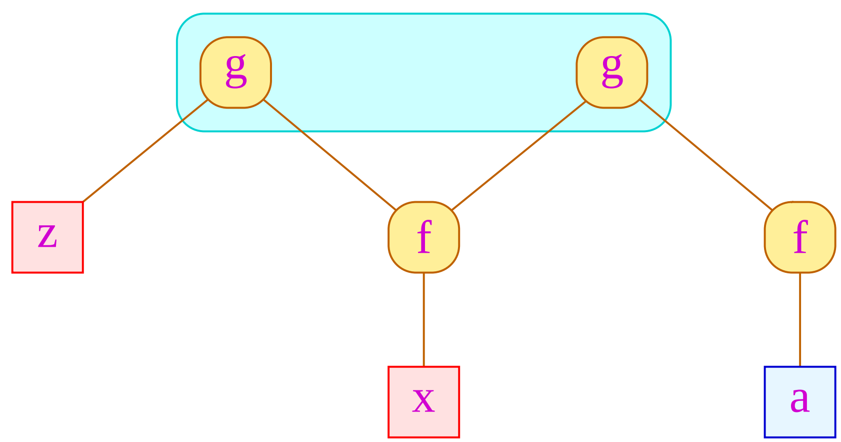


Idee:

- Wir repräsentieren die Terme der Gleichungen als Graphen.
- Dabei identifizieren wir bereits isomorphe Teilterme ;-)
- ...

... im Beispiel:

$$g(z, f(x)) = g(f(x), f(a))$$

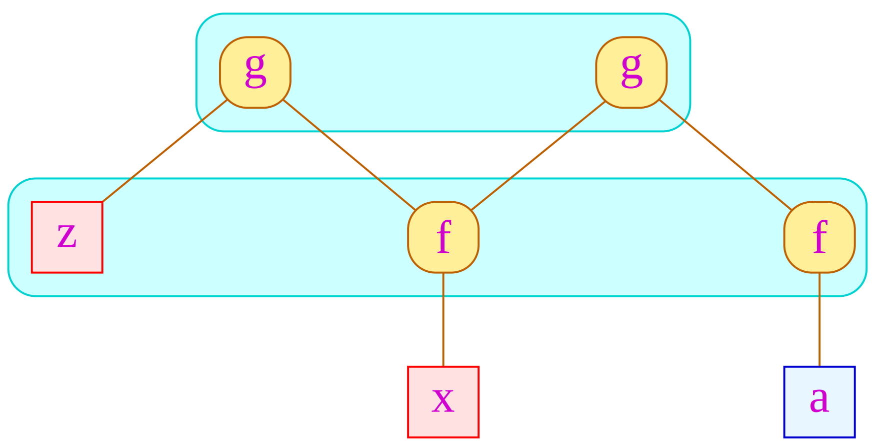


Idee:

- Wir repräsentieren die Terme der Gleichungen als Graphen.
- Dabei identifizieren wir bereits isomorphe Teilterme ;-)
- ...

... im Beispiel:

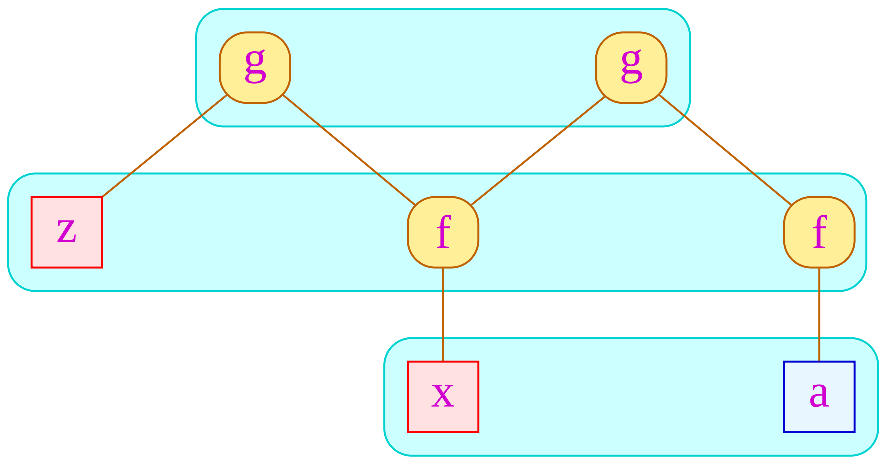
$$g(z, f(x)) \;=\; g(f(x), f(a))$$



Idee:

- Wir repräsentieren die Terme der Gleichungen als Graphen.
- Dabei identifizieren wir bereits isomorphe Teilterme ;-)
- ...

... im Beispiel: $g(z, f(x)) \;=\; g(f(x), f(a))$



Idee (Forts.):

- ...
- Wir berechnen eine Äquivalenz-Relation $\equiv$ auf den Knoten mit den folgenden Eigenschaften:

  $\longrightarrow$ $s \equiv t$ für jede Gleichung unseres Gleichungssystems;

  $\longrightarrow$ $s \equiv t$ nur, falls entweder s oder t eine Variable ist oder beide den gleichen Top-Konstruktor haben.

  $\longrightarrow$ Falls $s \equiv t$ und $s = f(s_1, \ldots, s_k), t = f(t_1, \ldots, t_k)$ dann auch $s_1 \equiv t_1, \ldots, s_k \equiv t_k$.

Idee (Forts.):

- ...
- Wir berechnen eine Äquivalenz-Relation $\equiv$ auf den Knoten mit den folgenden Eigenschaften:

  → $s \equiv t$ für jede Gleichung unseres Gleichungssystems;

  → $s \equiv t$ nur, falls entweder s oder t eine Variable ist oder beide den gleichen Top-Konstruktor haben.

  → Falls $s \equiv t$ und $s = f(s_1, \ldots, s_k), t = f(t_1, \ldots, t_k)$ dann auch $s_1 \equiv t_1, \ldots, s_k \equiv t_k$.

- Falls keine solche Äquivalenz-Relation existiert, ist das System unlösbar.
- Falls eine solche Äquivalenz-Relation gilt, müssen wir überprüfen, dass der Graph modulo der Äquivalenz-Relation azyklisch ist.
- Ist er azyklisch, können wir aus der Äquivalenzklasse jeder Variable eine allgemeinste Lösung ablesen ...

Implementierung:

- Wir verwalten eine Partition der Knoten;
- Wann immer zwei Knoten äquivalent sein sollen, vereinigen wir ihre Äquivalenzklassen und fahren mit den Söhnen entsprechend fort.
- Notwendige Operationen auf der Datenstruktur π für eine Partition:

  $\rightarrow$ init (Nodes) liefert eine Repräsentation für die Partition $\pi_0 = \{\{v\} \mid v \in \text{Nodes}\}$

  $\rightarrow$ find (π, u) liefert einen Repräsentanten der Äquivalenzklasse — der wann immer möglich keine Variable sein soll :-)

  $\rightarrow$ union (π, u_1, u_2) vereinigt die Äquivalenzklassen von u_1, u_2 :-)
- Der Algorithmus startet mit einer Liste

$$W = [(u_1, v_1), \ldots, (u_m, v_m)]$$

der Paare von Wurzelknoten der zu unifizierenden Terme ...

$$
\begin{array}{l}
\pi = \mathsf{init}(\mathsf{Nodes});\\
\textbf{while}\ \ (W \neq \emptyset)\ \ \{\\
\qquad (u, v) = \mathsf{Extract}\,(W);\\
\qquad u = \mathsf{find}\,(\pi, u);\ \ v = \mathsf{find}\,(\pi, v);\\
\qquad \textbf{if}\ \ (u \not\equiv v)\ \ \{\\
\qquad\qquad \pi = \mathsf{union}\,(\pi, u, v);\\
\qquad\qquad \textbf{if}\ \ (u \notin \mathsf{Vars} \wedge v \notin \mathsf{Vars})\\
\qquad\qquad\qquad \textbf{if}\ \ (\mathsf{label}(u) \neq \mathsf{label}(v))\ \ \textbf{return}\ \ \mathsf{Fail}\\
\qquad\qquad\qquad \textbf{else}\ \ \{\\
\qquad\qquad\qquad\qquad (u_1, \ldots, u_k) = \mathsf{Successors}(u);\\
\qquad\qquad\qquad\qquad (v_1, \ldots, v_k) = \mathsf{Successors}(v);\\
\qquad\qquad\qquad\qquad W = (u_1, v_1) :: \ldots :: (u_k, v_k) :: W;\\
\qquad\qquad\qquad \}\\
\qquad \}\\
\}
\end{array}
$$

Komplexität:

$\mathcal{O}(\#\, Knoten)$	Aufrufe von	union
$\mathcal{O}(\#\, Kanten + \#\, Gleichungen)$	Aufrufe von	find

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

Komplexität:

$\mathcal{O}(\#\, Knoten)$	Aufrufe von	union
$\mathcal{O}(\#\, Kanten + \#\, Gleichungen)$	Aufrufe von	find

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

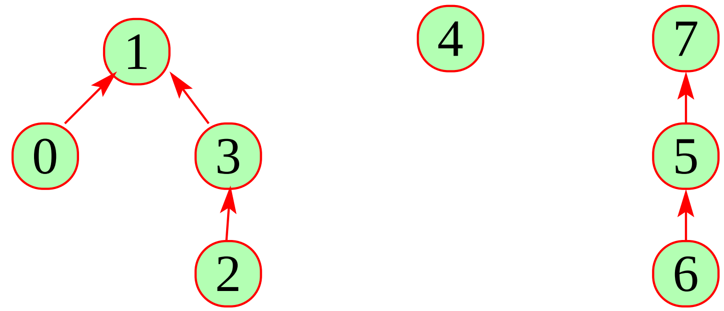
Idee:

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.

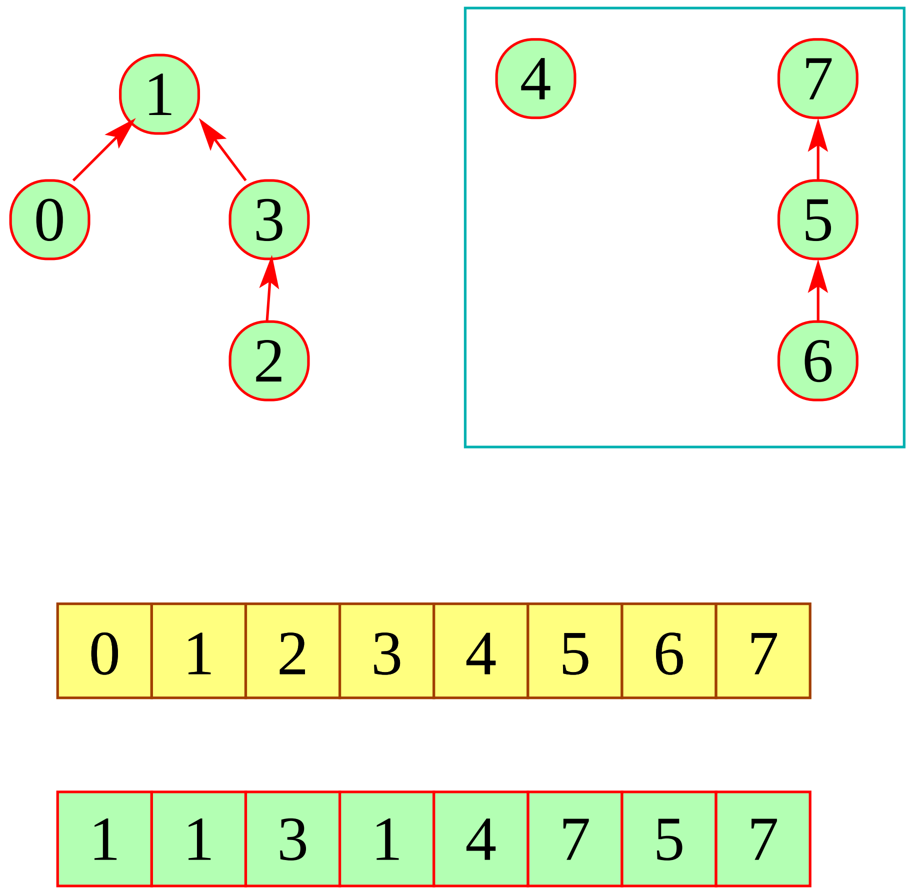
Ihre Wurzeln sind die Repräsentanten ...

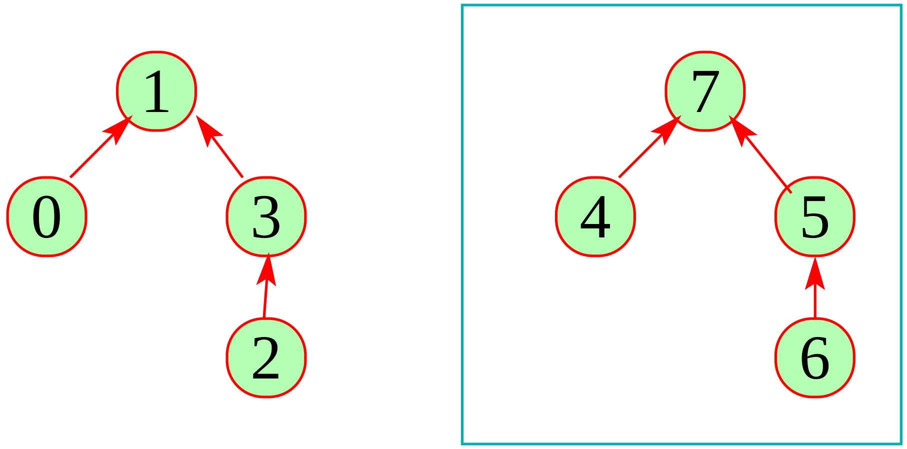


0	1	2	3	4	5	6	7
1	1	3	1	4	7	5	7

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

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


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



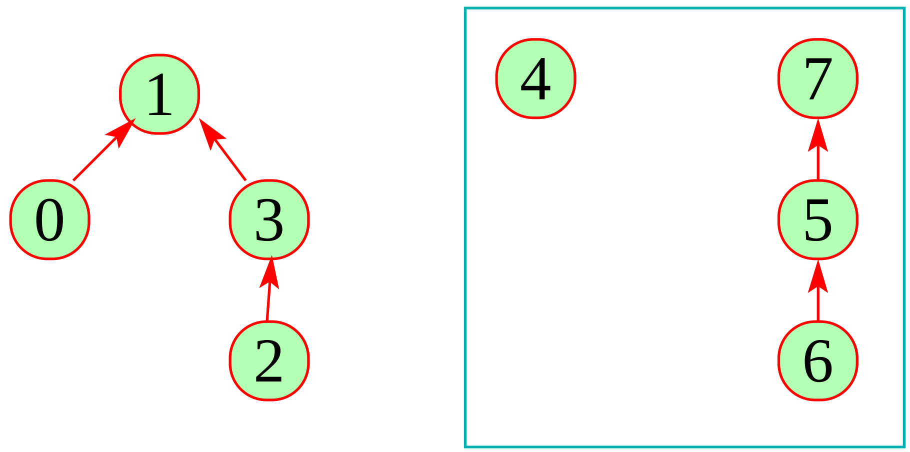
0	1	2	3	4	5	6	7
1	1	3	1	7	7	5	7

Die Kosten:

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

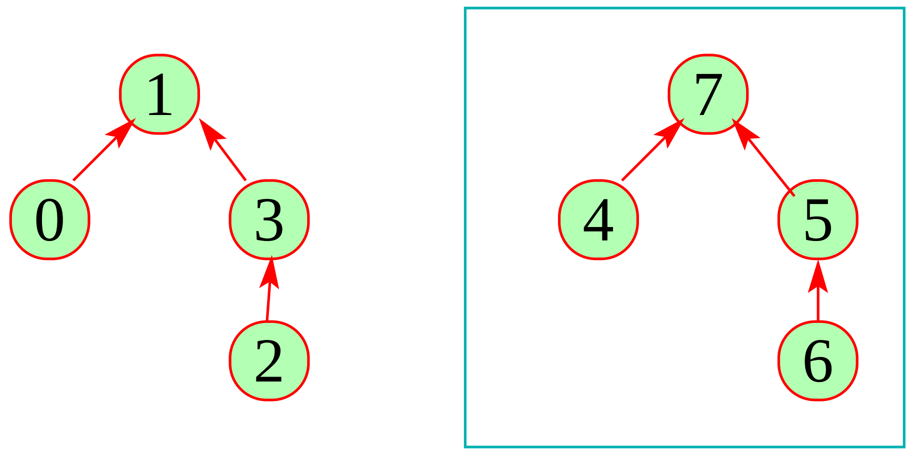
Strategie zur Vermeidung tiefer Bäume:

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



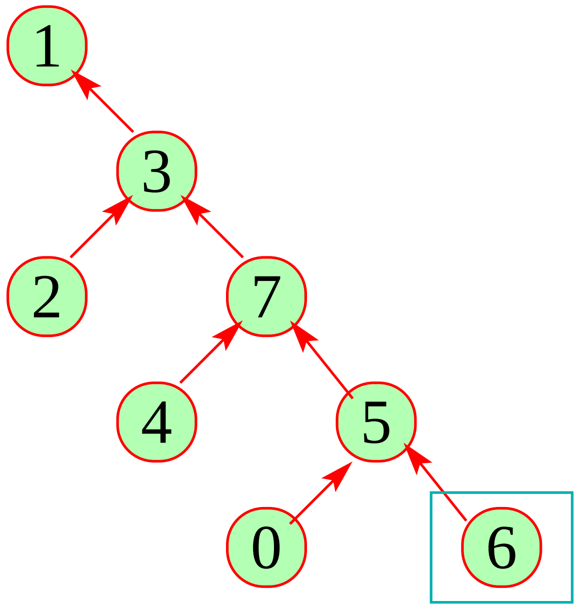
0	1	2	3	4	5	6	7

1	1	3	1	4	7	5	7

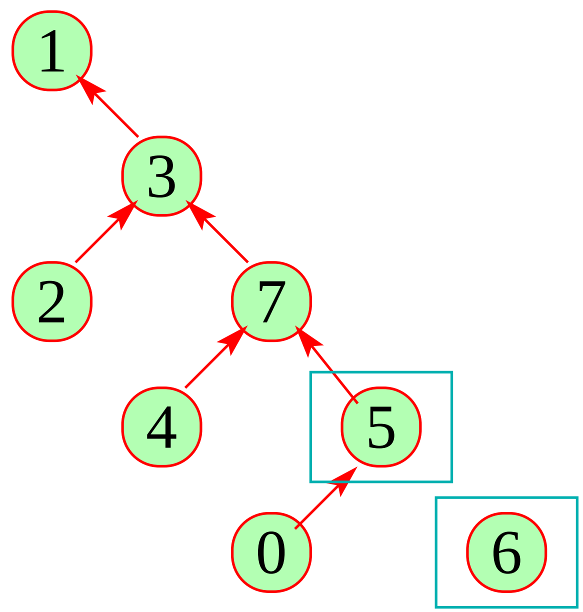


0	1	2	3	4	5	6	7

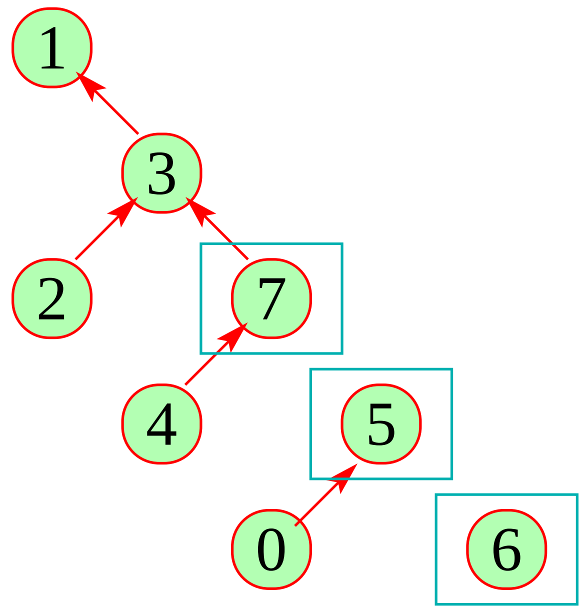
1	1	3	1	7	7	5	7



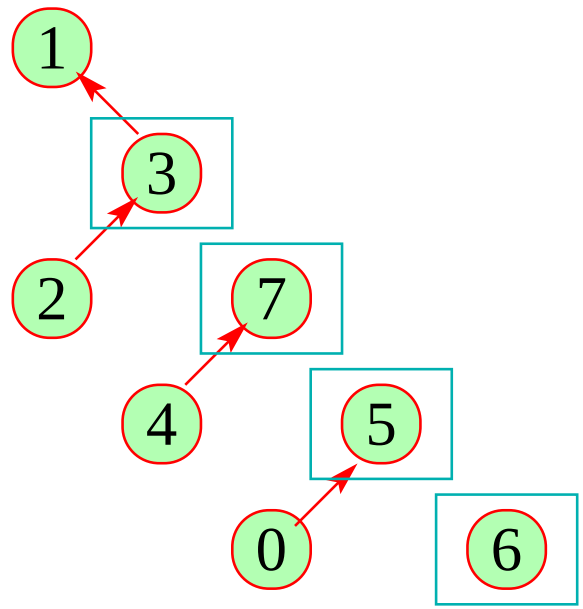
0	1	2	3	4	5	6	7
5	1	3	1	7	7	5	3



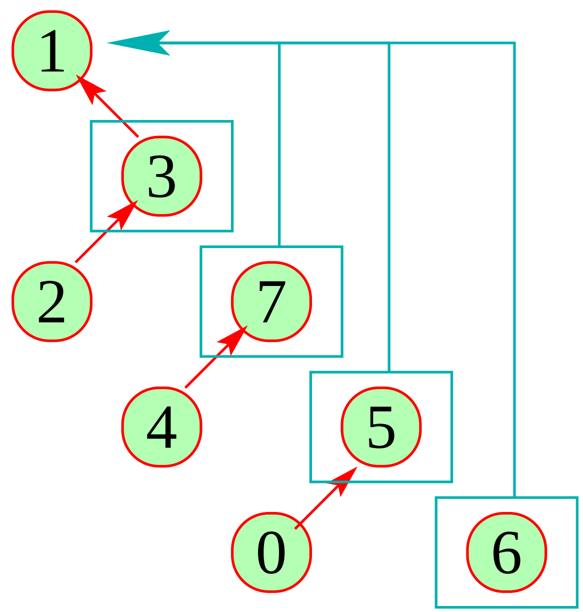
0	1	2	3	4	5	6	7
5	1	3	1	7	7	5	3



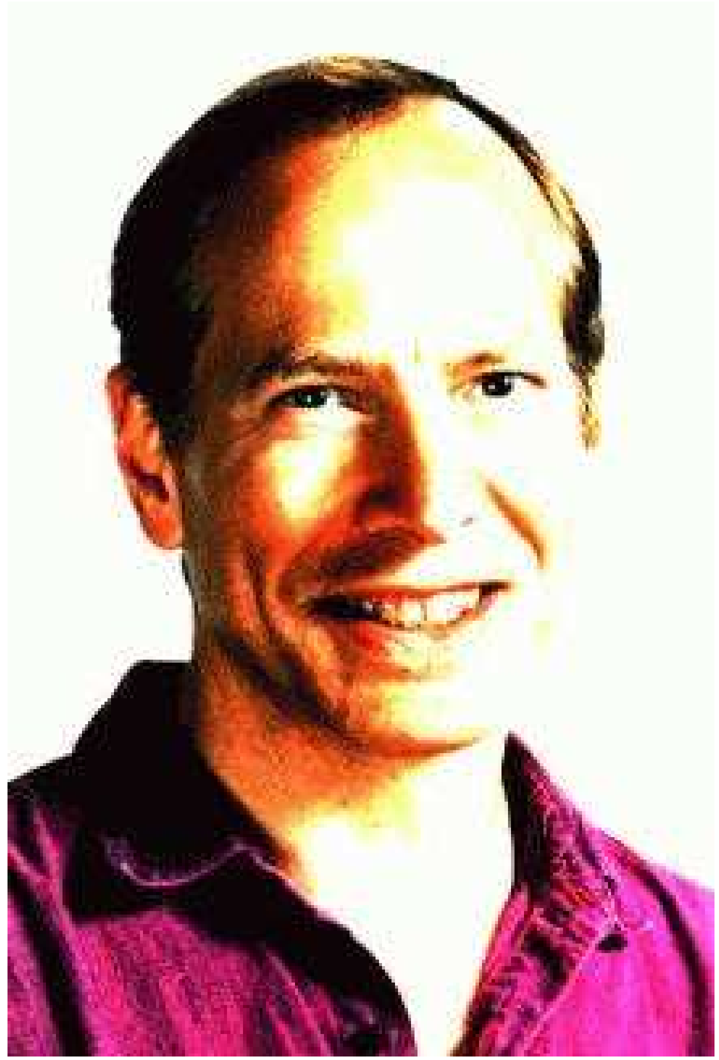
0	1	2	3	4	5	6	7
5	1	3	1	7	7	5	3



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

  // α die inverse Ackermann-Funktion :-)
- Für unsere Anwendung müssen wir union nur so modifizieren, dass an den Wurzeln nach Möglichkeit keine Variablen stehen.
- Diese Modifikation vergrößert die asymptotische Laufzeit nicht :-)

Fazit:

- Wenn Typ-Gleichungen für ein Programm lösbar sind, dann gibt es eine allgemeinste Zuordnung von Programm-Variablen und Teil-Ausdrücken zu Typen, die alle Regeln erfüllen :-)
- Eine solche allgemeinste Typisierung können wir in (fast) linearer Zeit berechnen :-)

Fazit:

- Wenn Typ-Gleichungen für ein Programm lösbar sind, dann gibt es eine allgemeinste Zuordnung von Programm-Variablen und Teil-Ausdrücken zu Typen, die alle Regeln erfüllen :-)
- Eine solche allgemeinste Typisierung können wir in (fast) linearer Zeit berechnen :-)

Achtung:

In der berechneten Typisierung können Typ-Variablen vorkommen !!!

Beispiel:

$$e \equiv \textbf{fn}\ (f, x) \Rightarrow f\ x$$

Mit $\alpha \equiv \alpha[x]$ und $\beta \equiv \tau[f\ x]$ finden wir:

$$\begin{array}{lll} \alpha[f] & = & \alpha \to \beta \\ \tau[e] & = & (\alpha \to \beta, \alpha) \to \beta \end{array}$$

Diskussion:

- Die Typ-Variablen bedeuten offenbar, dass die Funktionsdefinition für jede mögliche Instantiierung funktioniert $\Longrightarrow$ Polymorphie

  Wir kommen darauf zurück :-)
- Das bisherige Verfahren, um Typisierungen zu berechnen, hat den Nachteil, dass es nicht syntax-gerichtet ist ...
- Wenn das Gleichungssystem zu einem Programm keine Lösung besitzt, erhalten wir keine Information, wo der Fehler stecken könnte :-(

$\Longrightarrow$ Wir benötigen ein syntax-gerichtetes Verfahren !!!

... auch wenn es möglicherweise ineffizienter ist :-)