

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:
 - **init**(Nodes) liefert eine Repräsentation für die Partition
 $\pi_0 = \{\{v\} \mid v \in \text{Nodes}\}$
 - **find**(π, u) liefert einen Repräsentanten der Äquivalenzklasse —
der wann immer möglich keine Variable sein soll :-)
 - **union**(π, u_1, u_2) vereinigt die Äquivalenzklassen von u_1, u_2 :-)
- Der Algorithmus startet mit einer Liste

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

der Paare von Wurzelknoten der zu unifizierenden Terme ...

```

 $\pi = \text{init}(\text{Nodes});$ 
while ( $W \neq \emptyset$ ) {
     $(u, v) = \text{Extract}(W);$ 
     $u = \text{find}(\pi, u); v = \text{find}(\pi, v);$ 
    if ( $u \neq v$ ) {
         $\pi = \text{union}(\pi, u, v);$ 
        if ( $u \notin \text{Vars} \wedge v \notin \text{Vars}$ )
            if ( $\text{label}(u) \neq \text{label}(v)$ ) return Fail
            else {
                 $(u_1, \dots, u_k) = \text{Successors}(u);$ 
                 $(v_1, \dots, v_k) = \text{Successors}(v);$ 
                 $W = (u_1, v_1) :: \dots :: (u_k, v_k) :: W;$ 
            }
    }
}

```

Komplexität:

$\mathcal{O}(\# \text{ Knoten})$

Aufrufe von **union**

$\mathcal{O}(\# \text{ Kanten} + \# \text{ Gleichungen})$

Aufrufe von **find**

\implies Wir benötigen effiziente **Union-Find-Datenstruktur** :-)

Komplexität:

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

\implies Wir benötigen effiziente **Union-Find-Datenstruktur** :-)

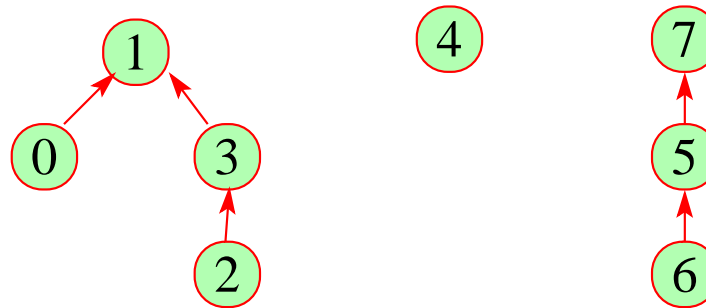
Idee:

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.

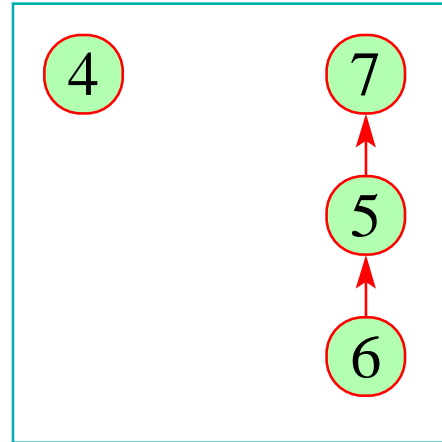
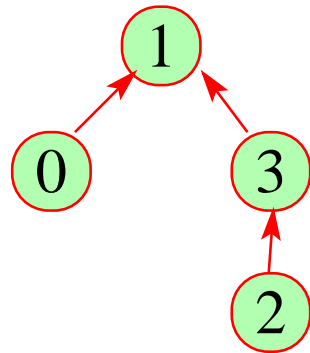
Ihre Wurzeln sind die Repräsentanten ...



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

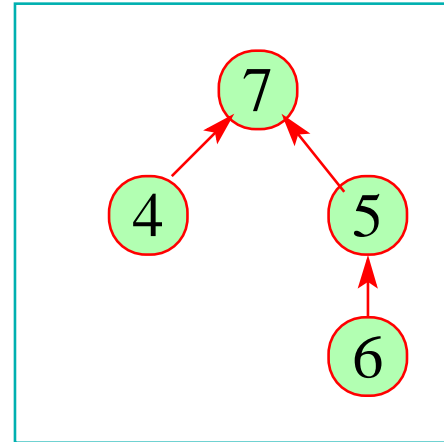
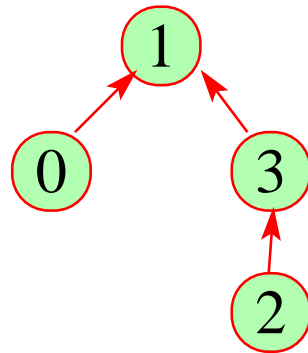
1	1	3	1	4	7	5	7
---	---	---	---	---	---	---	---

- **find** (π, u) folgt den Vater-Verweisen :-)
- **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
---	---	---	---	---	---	---	---

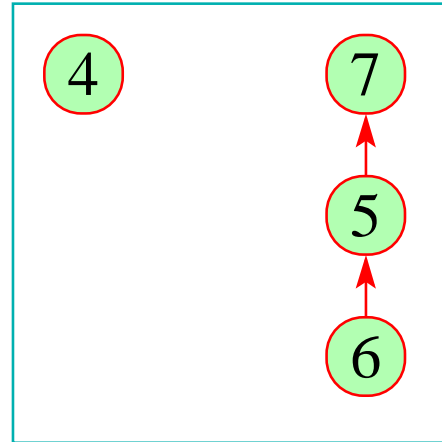
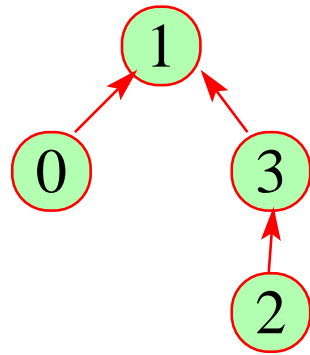
1	1	3	1	7	7	5	7
---	---	---	---	---	---	---	---

Die Kosten:

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

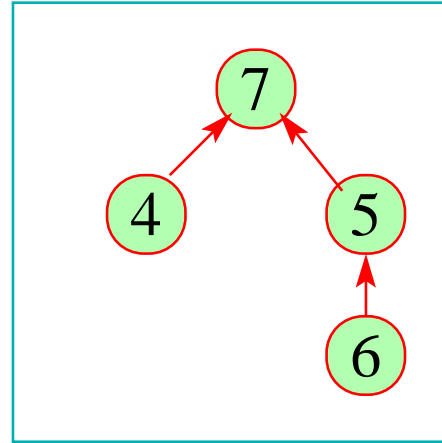
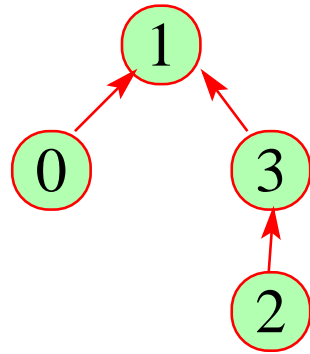
Strategie zur Vermeidung tiefer Bäume:

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



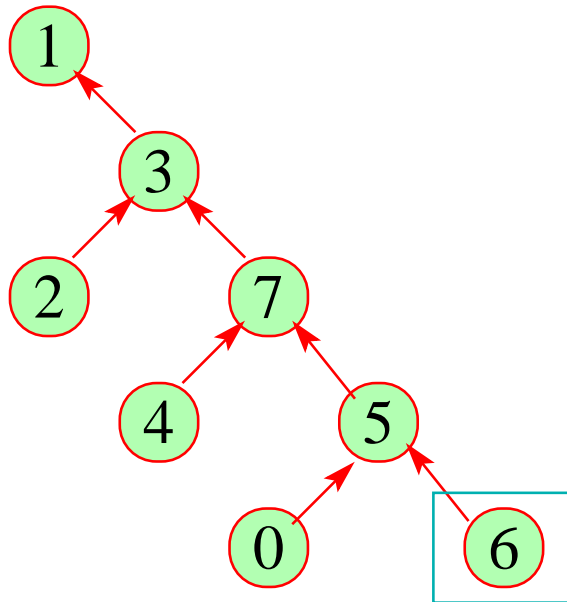
0	1	2	3	4	5	6	7
---	---	---	---	---	---	---	---

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

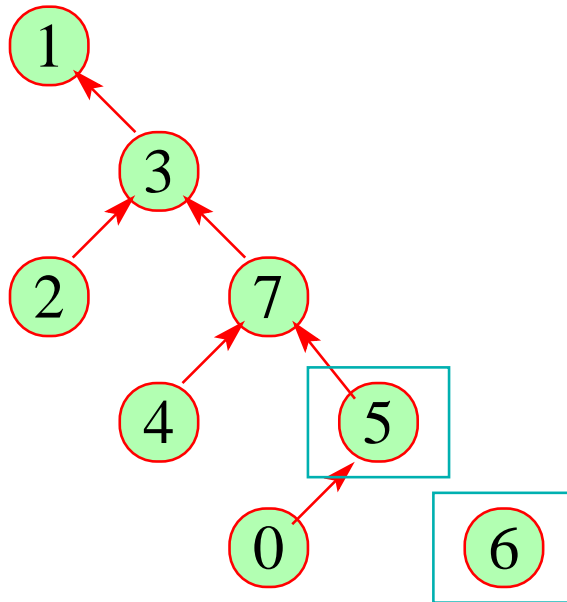


0	1	2	3	4	5	6	7
---	---	---	---	---	---	---	---

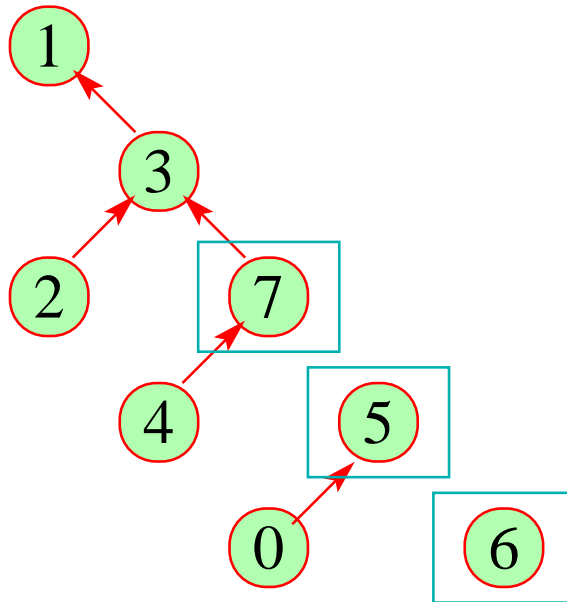
1	1	3	1	7	7	5	7
---	---	---	---	---	---	---	---



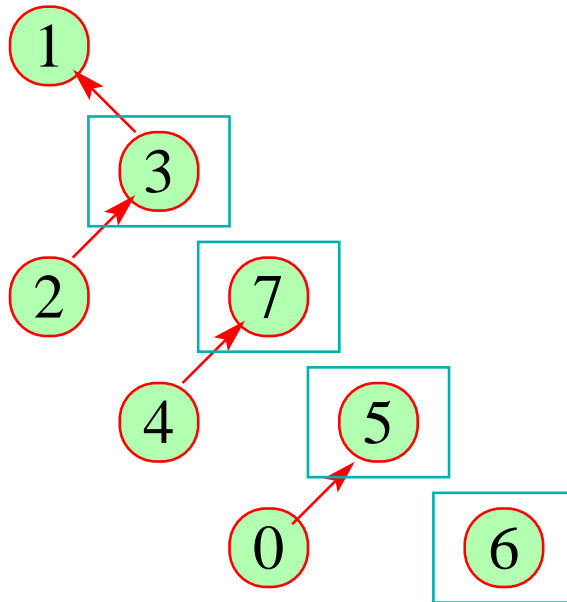
0	1	2	3	4	5	6	7
5	1	3	1	7	7	5	3



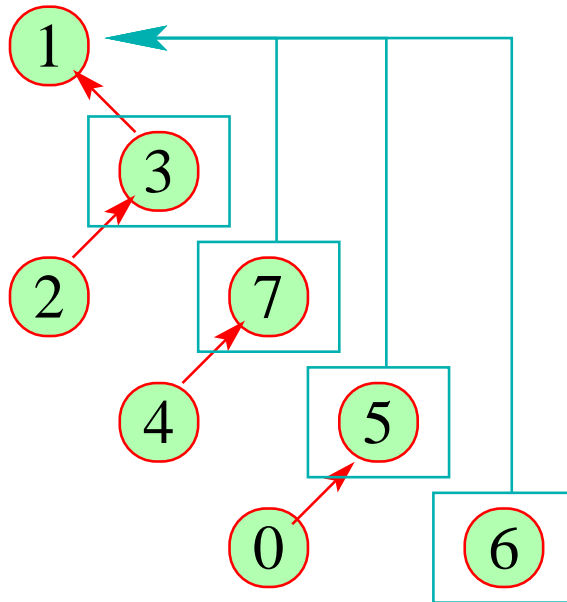
0	1	2	3	4	5	6	7
5	1	3	1	7	7	5	3



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** 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: $e \equiv \mathbf{fn} (f, x) \Rightarrow f x$

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

$$\alpha[f] = \alpha \rightarrow \beta$$

$$\tau[e] = (\alpha \rightarrow \beta, \alpha) \rightarrow \beta$$

Diskussion:

- Die Typ-Variablen bedeuten offenbar, dass die Funktionsdefinition für jede mögliche Instanziierung funktioniert \implies **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 :-)

\implies Wir benötigen ein syntax-gerichtetes Verfahren !!!
... auch wenn es möglicherweise ineffizienter ist :-)

Der Algorithmus \mathcal{W} :

```
fun  $\mathcal{W} e (\Gamma, \theta) = \text{case } e$ 
  of  $c$             $\rightarrow (t_c, \theta)$ 
    |  $[]$          $\rightarrow \text{let val } \alpha = \text{new}()$ 
       $\text{in } (\text{list } \alpha, \theta)$ 
       $\text{end}$ 
    |  $x$            $\rightarrow (\Gamma(x), \theta)$ 
    |  $(e_1, \dots, e_m)$   $\rightarrow \text{let val } (t_1, \theta) = \mathcal{W} e_1 (\Gamma, \theta)$ 
       $\dots$ 
       $\text{val } (t_m, \theta) = \mathcal{W} e_m (\Gamma, \theta)$ 
       $\text{in } ((t_1, \dots, t_m), \theta)$ 
       $\text{end}$ 
  ...
```

Der Algorithmus \mathcal{W} (Forts.):

```
|  $(e_1 : e_2)$   $\rightarrow$  let val  $(t_1, \theta) = \mathcal{W} e_1 (\Gamma, \theta)$   
    val  $(t_2, \theta) = \mathcal{W} e_2 (\Gamma, \theta)$   
    val  $\theta = \text{unify} (\text{list } t_1, t_2) \theta$   
in  $(t_2, \theta)$   
end  
  
|  $(e_1 e_2)$   $\rightarrow$  let val  $(t_1, \theta) = \mathcal{W} e_1 (\Gamma, \theta)$   
    val  $(t_2, \theta) = \mathcal{W} e_2 (\Gamma, \theta)$   
    val  $\alpha = \text{new} ()$   
    val  $\theta = \text{unify} (t_1, t_2 \rightarrow \alpha) \theta$   
in  $(\alpha, \theta)$   
end  
  
...
```

Der Algorithmus \mathcal{W} (Forts.):

```
| (if  $e_0$  then  $e_1$  else  $e_2$ ) → let val  $(t_0, \theta) = \mathcal{W} e_0 (\Gamma, \theta)$   
    val  $\theta = \text{unify}(\text{bool}, t_0) \theta$   
    val  $(t_1, \theta) = \mathcal{W} e_1 (\Gamma, \theta)$   
    val  $(t_2, \theta) = \mathcal{W} e_2 (\Gamma, \theta)$   
    val  $\theta = \text{unify}(t_1, t_2) \theta$   
in  $(t_1, \theta)$   
end
```

...

Der Algorithmus \mathcal{W} (Forts.):

```
| (case  $e_0$  of []  $\rightarrow e_1$ ;  $(x : y) \rightarrow e_2$ )  
   $\rightarrow$  let val  $(t_0, \theta) = \mathcal{W} e_0 (\Gamma, \theta)$   
        val  $\alpha = \text{new}()$   
        val  $\theta = \text{unify}(\text{list } \alpha, t_0) \theta$   
        val  $(t_1, \theta) = \mathcal{W} e_1 (\Gamma, \theta)$   
        val  $(t_2, \theta) = \mathcal{W} e_2 (\Gamma \oplus \{x \mapsto \alpha, y \mapsto \text{list } \alpha\}, \theta)$   
        val  $\theta = \text{unify}(t_1, t_2) \theta$   
  in  $(t_1, \theta)$   
  end
```

...

Der Algorithmus \mathcal{W} (Forts.):

```
| fn  $(x_1, \dots, x_m) \Rightarrow e$   
   $\rightarrow$  let val  $\alpha_1 = \text{new}()$   
       $\dots$   
      val  $\alpha_m = \text{new}()$   
      val  $(t, \theta) = \mathcal{W} e (\Gamma \oplus \{x_1 \mapsto \alpha_1, \dots, x_m \mapsto \alpha_m\}, \theta)$   
in  $((\alpha_1, \dots, \alpha_m) \rightarrow t, \theta)$   
end  
 $\dots$ 
```

Der Algorithmus \mathcal{W} (Forts.):

```
| (letrec  $x_1 = e_1; \dots; x_m = e_m$  in  $e_0$ )  
  → let val  $\alpha_1 = \text{new}()$   
      ...  
      val  $\alpha_m = \text{new}()$   
      val  $\Gamma = \Gamma \oplus \{x_1 \mapsto \alpha_1, \dots, x_m \mapsto \alpha_m\}$   
      val  $(t_1, \theta) = \mathcal{W} e_1 (\Gamma, \theta)$   
      val  $\theta = \text{unify}(\alpha_1, t_1) \theta$   
      ...  
      val  $(t_m, \theta) = \mathcal{W} e_m (\Gamma, \theta)$   
      val  $\theta = \text{unify}(\alpha_m, t_m) \theta$   
      val  $(t_0, \theta) = \mathcal{W} e_0 (\Gamma, \theta)$   
  in  $(t_0, \theta)$   
  end
```

...

Der Algorithmus \mathcal{W} (Forts.):

```
| (let  $x_1 = e_1; \dots; x_m = e_m$  in  $e_0$ )  
  → let val  $(t_1, \theta) = \mathcal{W} e_1 (\Gamma, \theta)$   
      val  $\Gamma = \Gamma \oplus \{x_1 \mapsto t_1\}$   
      ...  
      val  $(t_m, \theta) = \mathcal{W} e_m (\Gamma, \theta)$   
      val  $\Gamma = \Gamma \oplus \{x_m \mapsto t_m\}$   
      val  $(t_0, \theta) = \mathcal{W} e_0 (\Gamma, \theta)$   
  in  $(t_0, \theta)$   
  end
```

...

Bemerkungen:

- Am Anfang ist $\Gamma = \emptyset$ und $\theta = \emptyset$:-)
- Der Algorithmus unifiziert nach und nach die Typ-Gleichungen :-)
- Der Algorithmus liefert bei jedem Aufruf einen Typ t zusammen mit einer Substitution θ zurück.
- Der inferierte allgemeinste Typ ergibt sich als $\theta(t)$.
- Die Hilfsfunktion `new()` liefert jeweils eine neue Typvariable :-)
- Bei jedem Aufruf von `unify()` kann die Typinferenz **fehlschlagen** ...
- Bei Fehlschlag sollte die Stelle, wo der Fehler auftrat gemeldet werden, die Typ-Inferenz aber mit **plausiblen Werten** fortgesetzt werden :-}