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:

  $\longrightarrow$ $s \equiv t$ 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 $s_1 \equiv t_1, \ldots, s_k \equiv t_k$.

- 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:

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

  → $\text{find}\,(\pi, u)$ liefert einen Repräsentanten der Äquivalenzklasse — der wann immer möglich keine Variable sein soll :-)

  → $\text{union}\,(\pi, u_1, u_2)$ 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 ...

```
π = init(Nodes);
while (W ≠ ∅) {
        (u, v) = Extract (W);
        u = find (π, u);  v = find (π, v);
        if (u ≠ v) {
                π = union (π, u, v);
                if (u ∉ Vars ∧ v ∉ Vars)
                        if (label(u) ≠ label(v)) return Fail
                        else {
                                (u_1, ..., u_k) = Successors(u);
                                (v_1, ..., v_k) = Successors(u);
                                W = (u_1, v_1) :: ... :: (u_m, v_m) :: W;
                        }
        }
}
```

Komplexität:

$\mathcal{O}(\# \mathit{Knoten})$	Aufrufe von	union
$\mathcal{O}(\# \mathit{Kanten} + \# \mathit{Gleichungen})$	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 :-)

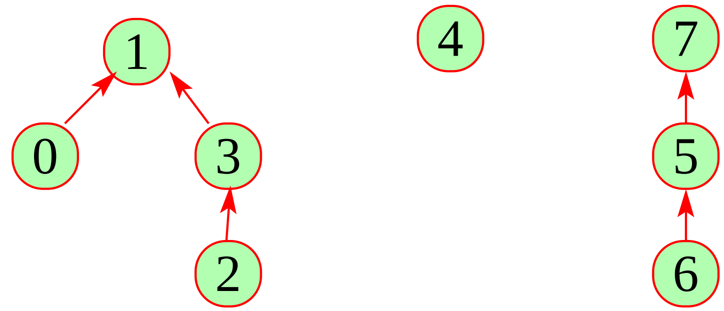
Idee:

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.

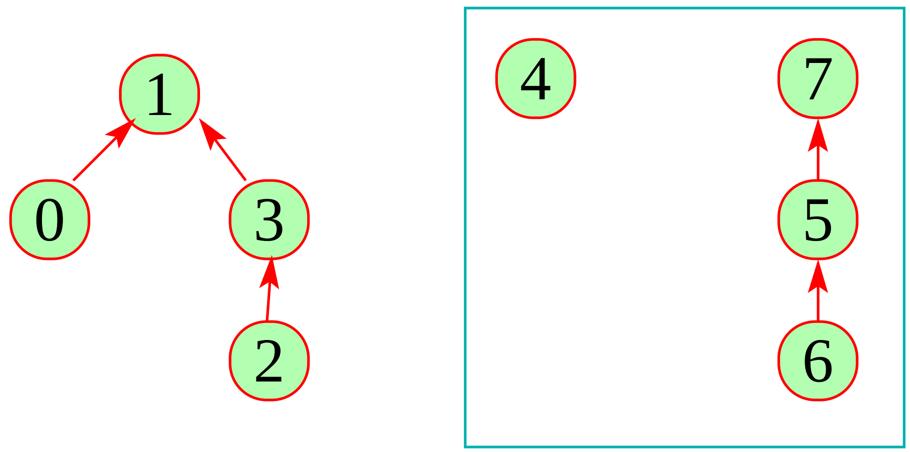
Ihre Wurzeln sind die Repräsentanten ...



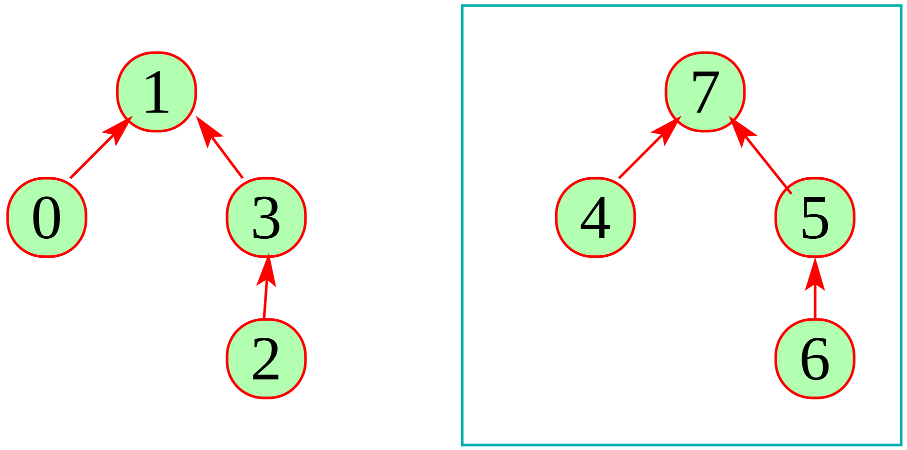
0	1	2	3	4	5	6	7
1	1	3	1	4	7	5	7

$\longrightarrow$ 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

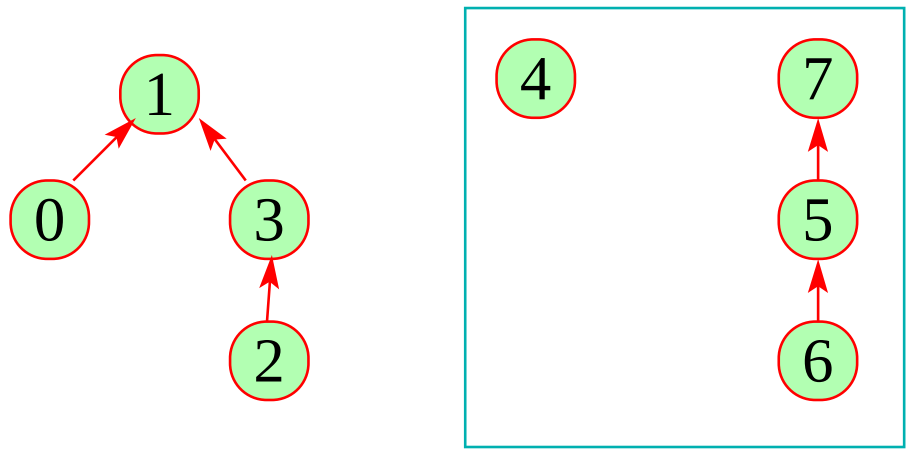
1	1	3	1	7	7	5	7

Die Kosten:

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

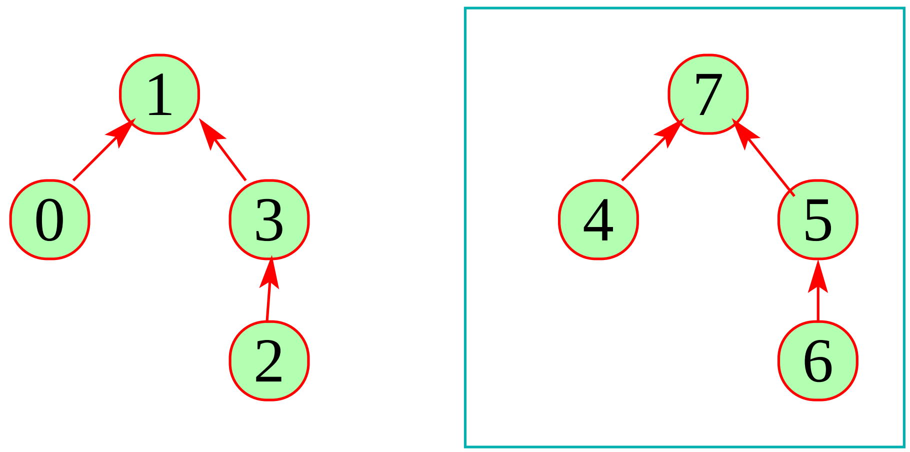
Strategie zur Vermeidung tiefer Bäume:

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



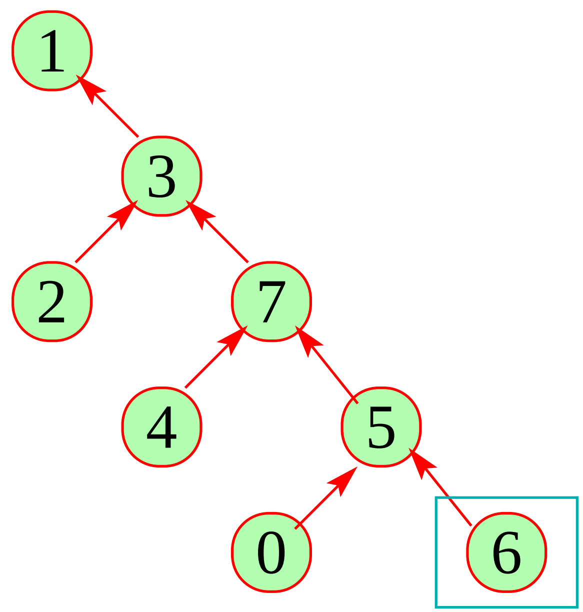
0	1	2	3	4	5	6	7

1	1	3	1	4	7	5	7

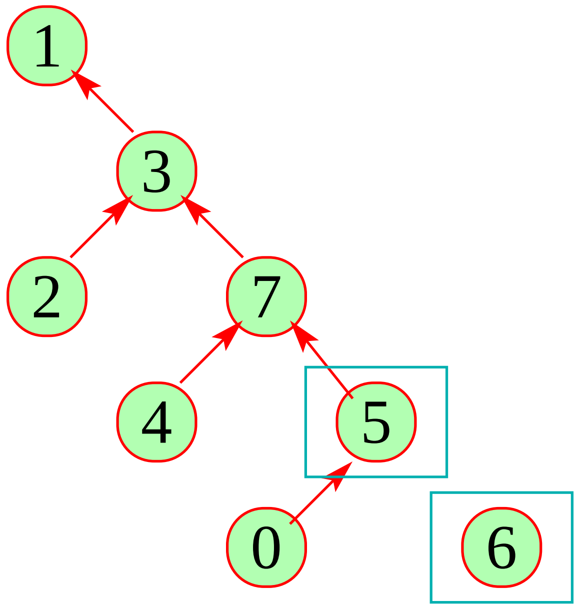


0	1	2	3	4	5	6	7

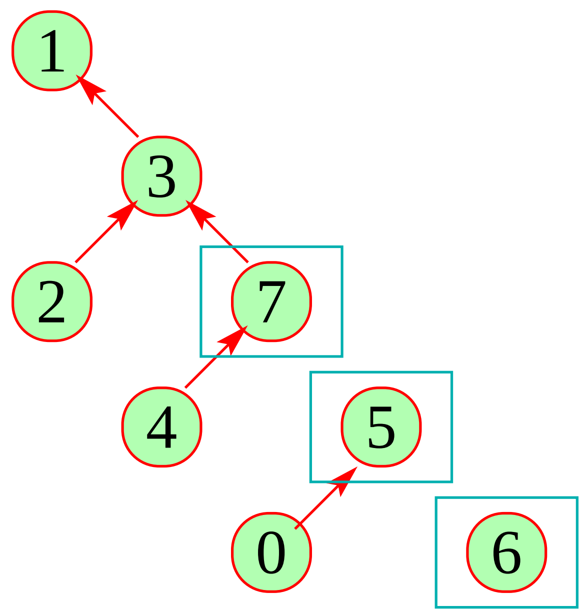
1	1	3	1	7	7	5	7



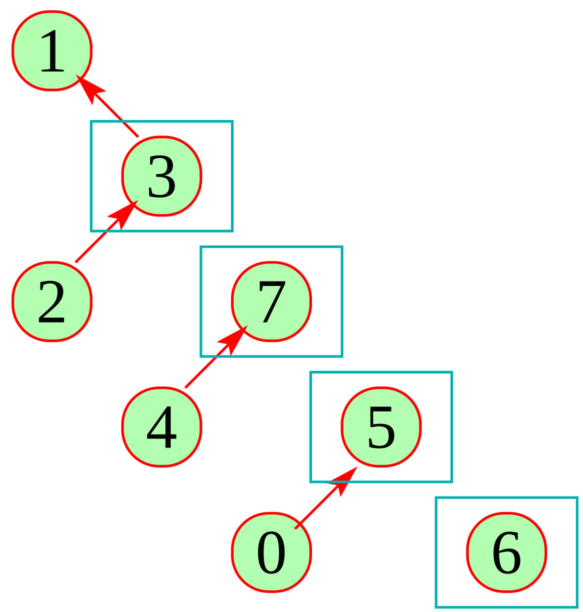
0	1	2	3	4	5	6	7
5	1	3	1	7	7	5	3



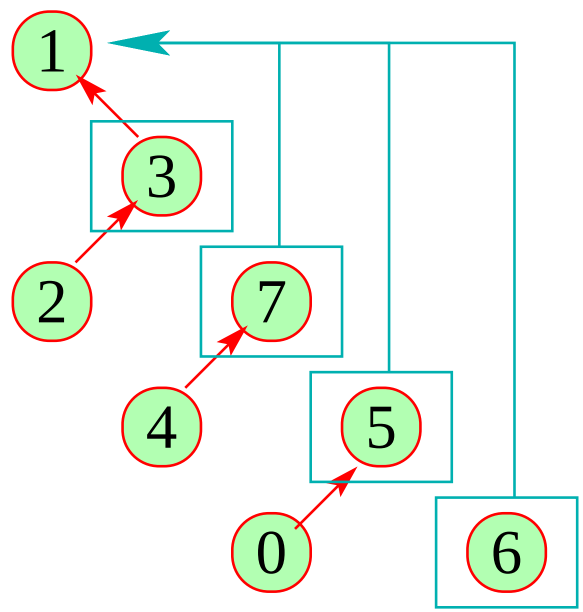
0	1	2	3	4	5	6	7
5	1	3	1	7	7	5	3



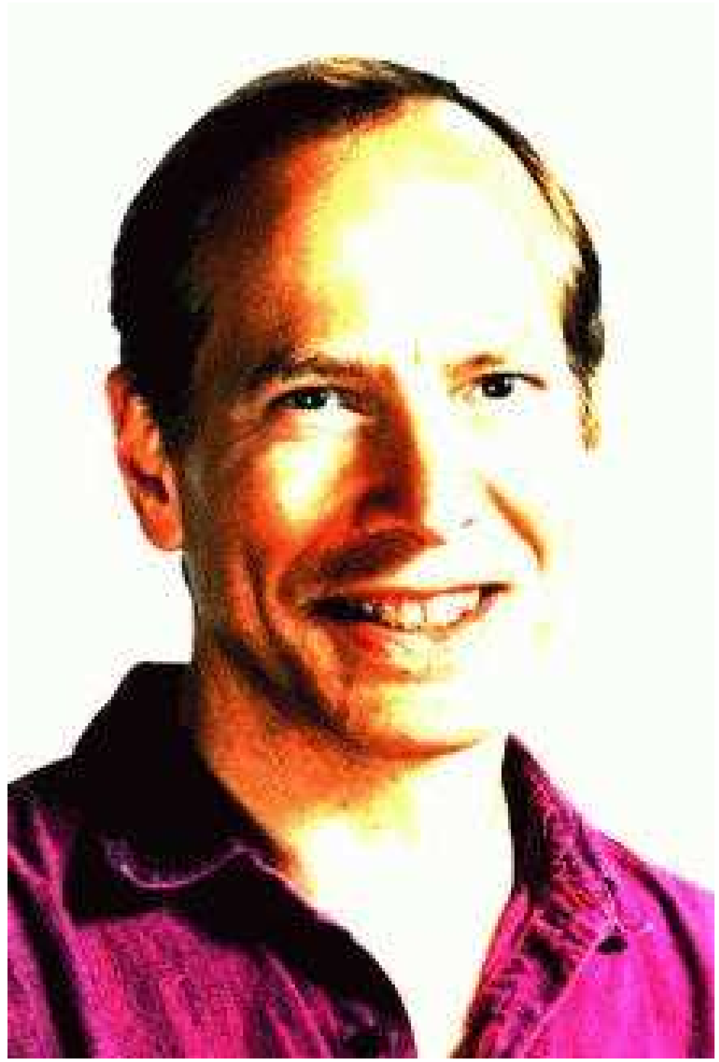
0	1	2	3	4	5	6	7
5	1	3	1	7	7	5	3



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 :-)

Der Algorithmus $\mathcal{W}$:

$$
\begin{array}{llll}
\textbf{fun } \mathcal{W}\ e\ (\Gamma, \theta) & = & \textbf{case } e & \\
\quad \textbf{of } c & & \rightarrow & (t_c, \theta) \\
\quad \mid\ [\,] & & \rightarrow & \textbf{let val } \alpha = \mathsf{new}() \\
 & & & \textbf{in } (\mathsf{list}\ \alpha, \theta) \\
 & & & \mathsf{end} \\
\quad \mid\ x & & \rightarrow & (\Gamma(x), \theta) \\
\quad \mid\ (e_1, \ldots, e_m) & & \rightarrow & \textbf{let val } (t_1, \theta) = \mathcal{W}\ e_1\ (\Gamma, \theta) \\
 & & & \qquad\qquad \ldots \\
 & & & \quad\ \textbf{val } (t_m, \theta) = \mathcal{W}\ e_m\ (\Gamma, \theta) \\
 & & & \textbf{in } ((t_1, \ldots, t_m), \theta) \\
 & & & \textbf{end} \\
 & & \ldots &
\end{array}
$$

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

$$
\begin{array}{llll}
\mid & (e_1 : e_2) & \to & \textbf{let val } (t_1, \theta) = \mathcal{W}\ e_1\ (\Gamma, \theta) \\
 & & & \phantom{\textbf{let }}\textbf{val } (t_2, \theta) = \mathcal{W}\ e_2\ (\Gamma, \theta) \\
 & & & \phantom{\textbf{let }}\textbf{val } \theta = \boxed{\mathsf{unify}\ (\mathsf{list}\ t_1, t_2)\ \theta} \\
 & & & \textbf{in } (t_2, \theta) \\
 & & & \textbf{end} \\
\mid & (e_1\ e_2) & \to & \textbf{let val } (t_1, \theta) = \mathcal{W}\ e_1\ (\Gamma, \theta) \\
 & & & \phantom{\textbf{let }}\textbf{val } (t_2, \theta) = \mathcal{W}\ e_2\ (\Gamma, \theta) \\
 & & & \phantom{\textbf{let }}\textbf{val } \alpha = \mathsf{new}\ () \\
 & & & \phantom{\textbf{let }}\textbf{val } \theta = \boxed{\mathsf{unify}\ (t_1, t_2 \to \alpha)\ \theta} \\
 & & & \textbf{in } (\alpha, \theta) \\
 & & & \textbf{end} \\
 & & \ldots &
\end{array}
$$

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

$$
\begin{array}{llll}
| & (\textbf{if}\ e_0\ \textbf{then}\ e_1\ \textbf{else}\ e_2) & \to & \textbf{let}\ \ \textbf{val}\ \ (t_0, \theta) = \mathcal{W}\ e_0\ (\Gamma, \theta) \\
& & & \qquad \textbf{val}\ \ \theta = \boxed{\mathsf{unify}\ (\textbf{bool}, t_0)\ \theta} \\
& & & \qquad \textbf{val}\ \ (t_1, \theta) = \mathcal{W}\ e_1\ (\Gamma, \theta) \\
& & & \qquad \textbf{val}\ \ (t_2, \theta) = \mathcal{W}\ e_2\ (\Gamma, \theta) \\
& & & \qquad \textbf{val}\ \ \theta = \boxed{\mathsf{unify}\ (t_1, t_2)\ \theta} \\
& & & \textbf{in}\ \ (t_1, \theta) \\
& & & \textbf{end}
\end{array}
$$

$$\ldots$$

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

$$
\begin{array}{ll}
| & (\textbf{case}\ e_0\ \textbf{of}\ [\,] \to e_1\ ;\ (x:y) \to e_2) \\
& \quad \to \quad \textbf{let}\ \ \textbf{val}\ \ (t_0, \theta) = \mathcal{W}\ e_0\ (\Gamma, \theta) \\
& \qquad\qquad\quad \textbf{val}\ \ \alpha = \mathsf{new}() \\
& \qquad\qquad\quad \textbf{val}\ \ \theta = \boxed{\mathsf{unify}\ (\mathsf{list}\ \alpha, t_0)\ \theta} \\
& \qquad\qquad\quad \textbf{val}\ \ (t_1, \theta) = \mathcal{W}\ e_1\ (\Gamma, \theta) \\
& \qquad\qquad\quad \textbf{val}\ \ (t_2, \theta) = \mathcal{W}\ e_2\ (\Gamma \oplus \{x \mapsto \alpha, y \mapsto \mathsf{list}\ \alpha\}, \theta) \\
& \qquad\qquad\quad \textbf{val}\ \ \theta = \boxed{\mathsf{unify}\ (t_1, t_2)\ \theta} \\
& \qquad\quad\ \ \textbf{in}\ \ (t_1, \theta) \\
& \qquad\quad\ \ \textbf{end} \\
& \ldots
\end{array}
$$

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

$$
\begin{array}{llll}
\mid & \textbf{fn}\ (x_1,\ldots,x_m)\ \Rightarrow\ e & & \\
 & \quad\rightarrow & \textbf{let}\ \ \textbf{val}\ \ \alpha_1 = \mathsf{new}() & \\
 & & \qquad\qquad \ldots & \\
 & & \qquad\ \textbf{val}\ \ \alpha_m = \mathsf{new}() & \\
 & & \qquad\ \textbf{val}\ \ (t,\theta) = \mathcal{W}\ e\ (\Gamma \oplus \{x_1 \mapsto \alpha_1,\ldots,x_m \mapsto \alpha_m\},\theta) & \\
 & & \textbf{in}\ \ ((\alpha_1,\ldots,\alpha_m) \rightarrow t,\theta) & \\
 & & \textbf{end} & \\
 & \ldots & &
\end{array}
$$

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

$$
\begin{array}{lll}
| & (\textbf{letrec}\ x_1 = e_1; \ldots; x_m = e_m\ \textbf{in}\ e_0) & \\
& \quad \to \quad \textbf{let}\ \ \textbf{val}\ \ \alpha_1 = \mathsf{new}() & \\
& \qquad\qquad\qquad \ldots & \\
& \qquad\qquad\quad \textbf{val}\ \ \alpha_m = \mathsf{new}() & \\
& \qquad\qquad\quad \textbf{val}\ \ \Gamma = \Gamma \oplus \{x_1 \mapsto \alpha_1, \ldots, x_m \mapsto \alpha_m\} & \\
& \qquad\qquad\quad \textbf{val}\ \ (t_1, \theta) = \mathcal{W}\ e_1\ (\Gamma, \theta) & \\
& \qquad\qquad\quad \textbf{val}\ \ \theta = \boxed{\mathsf{unify}\ (\alpha_1, t_1)\ \theta} & \\
& \qquad\qquad\qquad \ldots & \\
& \qquad\qquad\quad \textbf{val}\ \ (t_m, \theta) = \mathcal{W}\ e_m\ (\Gamma, \theta) & \\
& \qquad\qquad\quad \textbf{val}\ \ \theta = \boxed{\mathsf{unify}\ (\alpha_m, t_m)\ \theta} & \\
& \qquad\qquad\quad \textbf{val}\ \ (t_0, \theta) = \mathcal{W}\ e_0\ (\Gamma, \theta) & \\
& \qquad\quad\ \ \textbf{in}\ \ (t_0, \theta) & \\
& \qquad\quad\ \ \textbf{end} &
\end{array}
$$

. . .

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

$$
\begin{array}{lll}
| & (\textbf{let } x_1 = e_1; \ldots; x_m = e_m \textbf{ in } e_0) & \\
 & \to & \textbf{let val } (t_1, \theta) = \mathcal{W}\ e_1\ (\Gamma, \theta) \\
 & & \quad \textbf{val } \Gamma = \Gamma \oplus \{x_1 \mapsto t_1\} \\
 & & \quad\quad \ldots \\
 & & \quad \textbf{val } (t_m, \theta) = \mathcal{W}\ e_m\ (\Gamma, \theta) \\
 & & \quad \textbf{val } \Gamma = \Gamma \oplus \{x_m \mapsto t_m\} \\
 & & \quad \textbf{val } (t_0, \theta) = \mathcal{W}\ e_0\ (\Gamma, \theta) \\
 & & \textbf{in } (t_0, \theta) \\
 & & \textbf{end} \\
 & \ldots &
\end{array}
$$

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 $\mathsf{new}()$ liefert jeweils eine neue Typvariable :-)
- Bei jedem Aufruf von $\mathsf{unify}()$ kann die Typinferenz fehlschlagen ...
- Bei Fehlschlag sollte die Stelle, wo der Fehler auftrat gemeldet werden, die Typ-Inferenz aber mit plausiblen Werten fortgesetzt werden :-}

Beispiel:

$$
\begin{array}{lllllllll}
\textbf{let} & \mathsf{dup} & = & \textbf{fn}\ f & \Rightarrow & \textbf{fn}\ x & \Rightarrow & f\,(f\,x); \\
 & \mathsf{inc} & = & \textbf{fn}\ y & \Rightarrow & y+1; & & \\
 & \mathsf{single} & = & \textbf{fn}\ y & \Rightarrow & y : [\,] & & \\
\multicolumn{9}{l}{\textbf{in}\ \ \mathsf{dup\ single}\ (\mathsf{dup\ inc}\ 1)} \\
\multicolumn{9}{l}{\textbf{end}}
\end{array}
$$

Beispiel:

$$
\begin{array}{llcl}
\textbf{let} & \mathsf{dup} & = & \textbf{fn}\ f \Rightarrow \textbf{fn}\ x \Rightarrow f\,(f\,x); \\
 & \mathsf{inc} & = & \textbf{fn}\ y \Rightarrow y + 1; \\
 & \mathsf{single} & = & \textbf{fn}\ y \Rightarrow y : [\,] \\
\textbf{in} & \multicolumn{3}{l}{\mathsf{dup}\ \mathsf{single}\ (\mathsf{dup}\ \mathsf{inc}\ 1)} \\
\textbf{end} & & &
\end{array}
$$

Wir finden:

$$
\begin{array}{lcl}
\alpha[\mathsf{dup}] & = & (\alpha \to \beta) \to \alpha \to \beta \\
\alpha[\mathsf{inc}] & = & \textbf{int} \to \textbf{int} \\
\alpha[\mathsf{single}] & = & \gamma \to \mathsf{list}\ \gamma
\end{array}
$$

- Durch die Anwendung: dup single erhalten wir:

$$
\begin{array}{lll}
\alpha & = & \gamma \\
\beta & = & \text{list } \gamma \\
\alpha[\text{dup}] & = & (\gamma \to \text{list } \gamma) \to \gamma \to \text{list } \gamma
\end{array}
$$

- Durch die Anwendung: dup inc erhalten wir:

$$
\begin{array}{lll}
\alpha & = & \mathbf{int} \\
\beta & = & \mathbf{int} \\
\alpha[\text{dup}] & = & (\mathbf{int} \to \mathbf{int}) \to \mathbf{int} \to \mathbf{int}
\end{array}
$$

$\Longrightarrow$ Typ-Fehler ???

Idee 1: Kopiere jede Definition für jede Benutzung ...

... im Beispiel:

$$
\begin{array}{lllllll}
\textbf{let} & \mathsf{dup} & = & \textbf{fn}\ f & \Rightarrow & \textbf{fn}\ x & \Rightarrow\ f\,(f\,x); \\
& \mathsf{inc} & = & \textbf{fn}\ y & \Rightarrow & y+1; & \\
& \mathsf{single} & = & \textbf{fn}\ y & \Rightarrow & y : [\,] & \\
\textbf{in} & \multicolumn{6}{l}{(\textbf{fn}\ f \ \Rightarrow\ \textbf{fn}\ x\ \Rightarrow\ f\,(f\,x))\ \mathsf{single}} \\
& \multicolumn{6}{l}{((\textbf{fn}\ f \ \Rightarrow\ \textbf{fn}\ x\ \Rightarrow\ f\,(f\,x))\ \mathsf{inc}\ 1)} \\
\textbf{end} & & & & & &
\end{array}
$$

Idee 1: Kopiere jede Definition für jede Benutzung ...

... im Beispiel:

$$
\begin{array}{lllllll}
\textbf{let} & \textsf{dup} & = & \textbf{fn}\ f \Rightarrow \textbf{fn}\ x \Rightarrow f\,(f\,x); \\
 & \textsf{inc} & = & \textbf{fn}\ y \Rightarrow y+1; \\
 & \textsf{single} & = & \textbf{fn}\ y \Rightarrow y : [\,] \\
\textbf{in} & \multicolumn{3}{l}{(\textbf{fn}\ f \Rightarrow \textbf{fn}\ x \Rightarrow f\,(f\,x))\ \textsf{single}} \\
 & \multicolumn{3}{l}{((\textbf{fn}\ f \Rightarrow \textbf{fn}\ x \Rightarrow f\,(f\,x))\ \textsf{inc}\ 1)} \\
\textbf{end}
\end{array}
$$

- \+ Die beiden Teilausdrücke $(\textbf{fn}\ f \Rightarrow \textbf{fn}\ x \Rightarrow f\,(f\,x))$ erhalten jeweils einen eigenen Typ mit unabhängigen Typ-Variablen :-)
- \+ Das expandierte Programm ist typbar :-))
- – Das expandierte Programm kann seeehr groß werden :-(
- – Typ-Checking ist nicht mehr modular :-((