Komplexität:

$\mathcal{O}(\#\,Knoten)$ Aufrufe von union

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

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

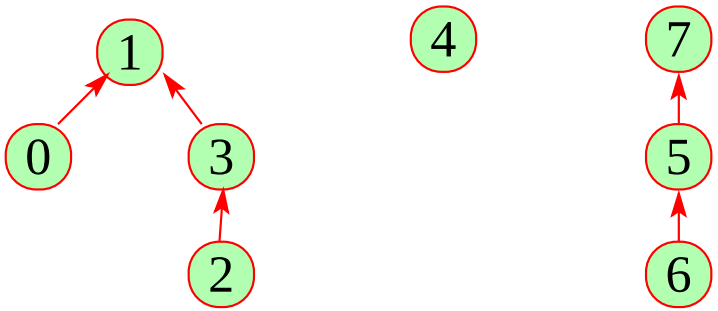
Idee:

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 ...

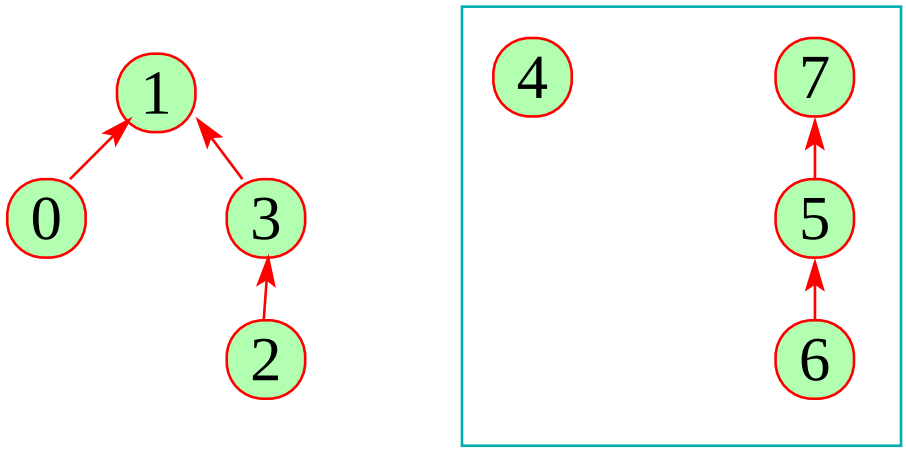


0	1	2	3	4	5	6	7

1	1	3	1	4	7	5	7

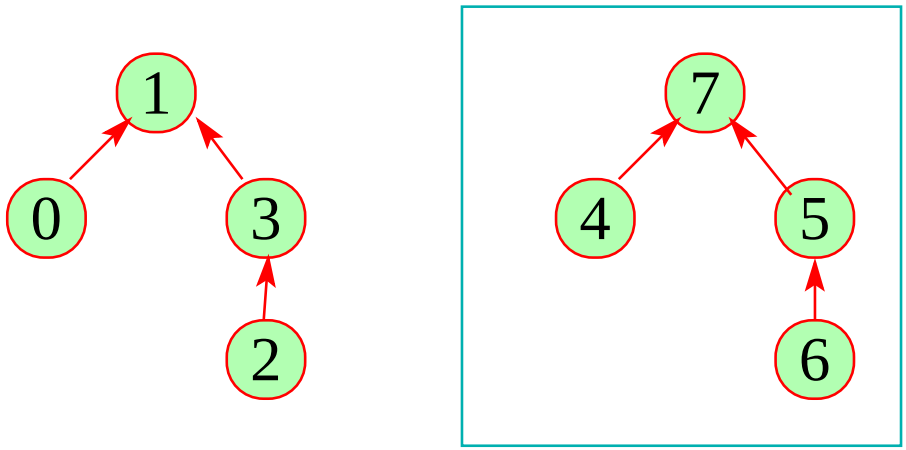
$\rightarrow$ find (π, u) folgt den Vater-Verweisen :-)

$\rightarrow$ 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

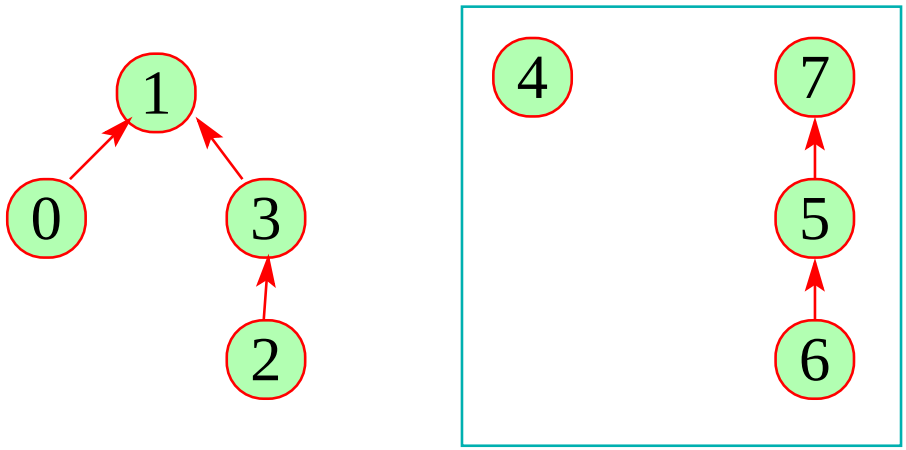
1	1	3	1	7	7	5	7

Die Kosten:

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

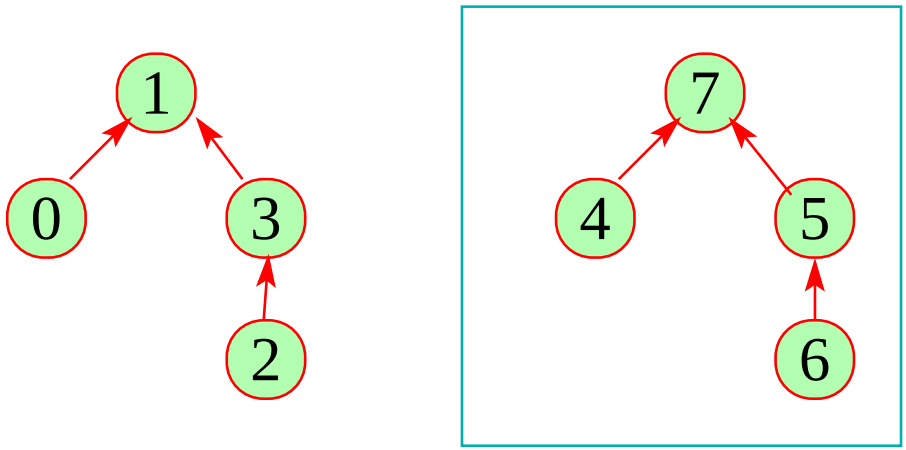
Strategie zur Vermeidung tiefer Bäume:

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



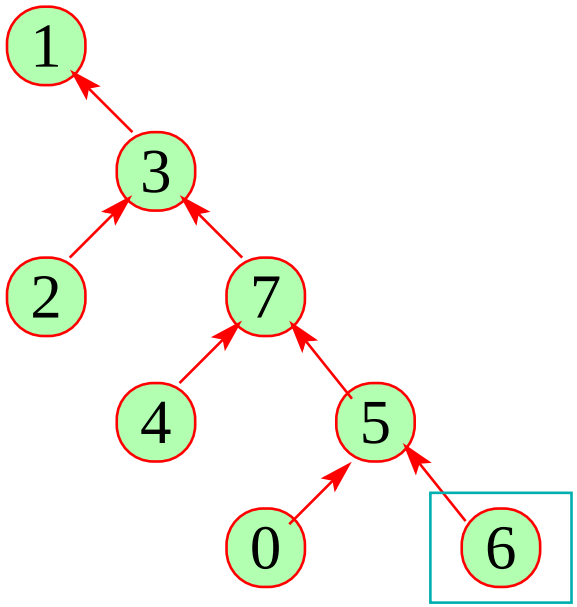
0	1	2	3	4	5	6	7

1	1	3	1	4	7	5	7

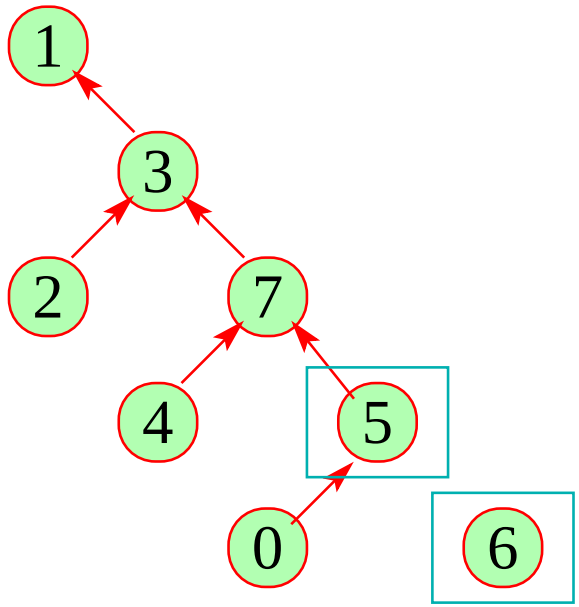


0	1	2	3	4	5	6	7

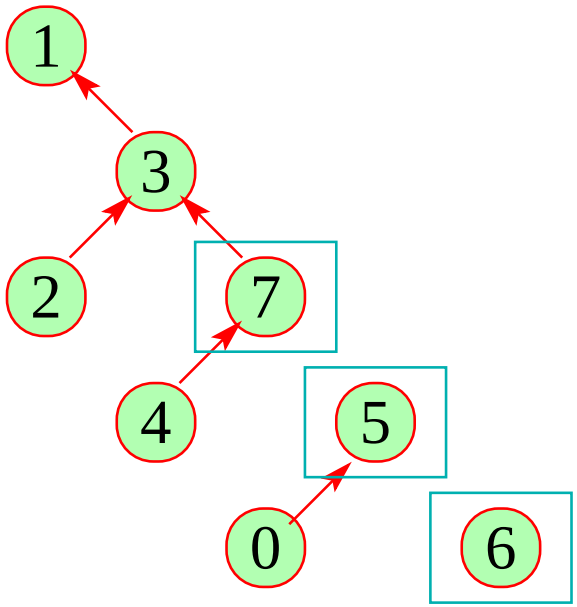
1	1	3	1	7	7	5	7



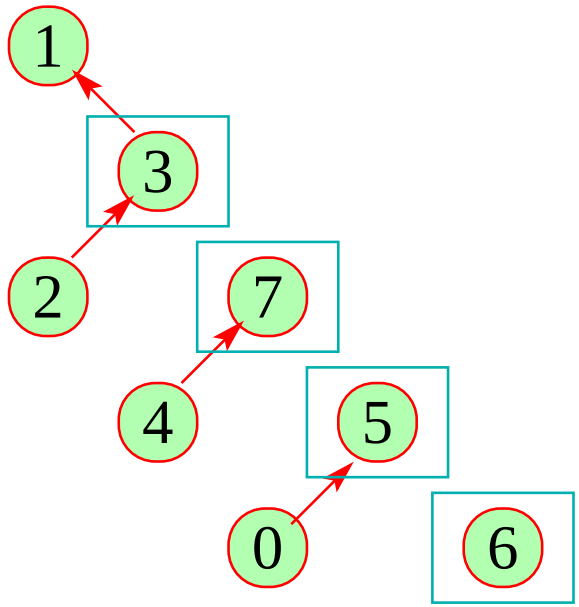
0	1	2	3	4	5	6	7
5	1	3	1	7	7	5	3



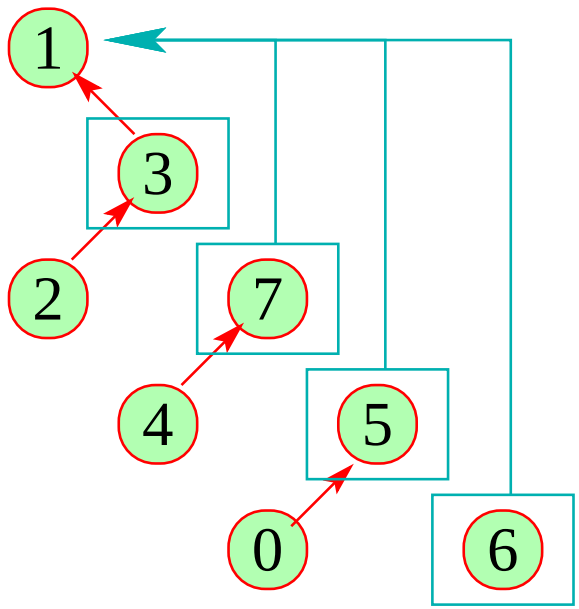
0	1	2	3	4	5	6	7
5	1	3	1	7	7	5	3



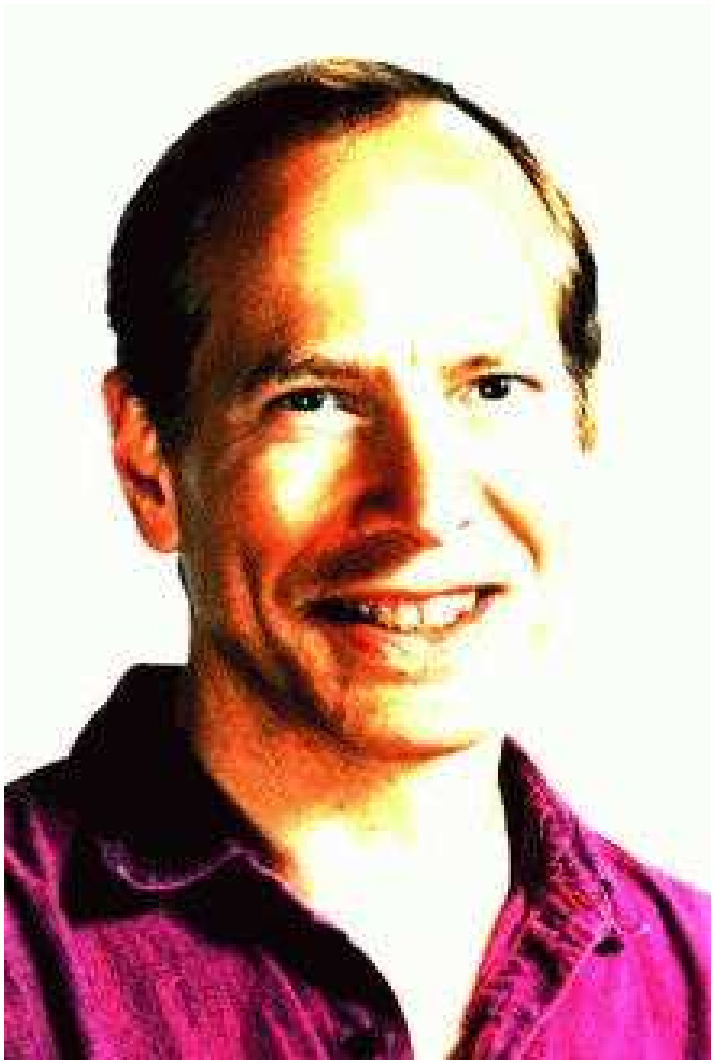
0	1	2	3	4	5	6	7
5	1	3	1	7	7	5	3



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

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

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

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 $\mathcal{W}$:

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

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

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

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

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

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

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

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

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

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

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

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

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

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 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 :-}

Beispiel:

$$
\begin{array}{l}
\textbf{let}\ \mathsf{apply} \quad = \quad \textbf{fn}\ f \ \Rightarrow\ \textbf{fn}\ x \ \Rightarrow\ f\ x; \\
\qquad \mathsf{inc} \quad = \quad \textbf{fn}\ y \ \Rightarrow\ y + 1; \\
\qquad \mathsf{single} \quad = \quad \textbf{fn}\ y \ \Rightarrow\ y : [\,] \\
\textbf{in}\ \ \mathsf{apply}\ \mathsf{single}\ (\mathsf{apply}\ \mathsf{inc}\ 1) \\
\textbf{end}
\end{array}
$$

Beispiel:

$$
\begin{array}{lll}
\textbf{let}\ \mathsf{apply} & = & \textbf{fn}\ f\ \Rightarrow\ \textbf{fn}\ x\ \Rightarrow\ f\ x; \\
\quad \mathsf{inc} & = & \textbf{fn}\ y\ \Rightarrow\ y + 1; \\
\quad \mathsf{single} & = & \textbf{fn}\ y\ \Rightarrow\ y : [\,] \\
\textbf{in}\ \mathsf{apply}\ \mathsf{single}\ (\mathsf{apply}\ \mathsf{inc}\ 1) & & \\
\textbf{end} & &
\end{array}
$$

Wir finden:

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

- Durch die Anwendung: apply single erhalten wir:

$$\begin{aligned} \alpha &= \gamma \\ \beta &= \mathsf{list}\ \gamma \\ \alpha[\mathsf{apply}] &= (\gamma \to \mathsf{list}\ \gamma) \to \gamma \to \mathsf{list}\ \gamma \end{aligned}$$

- Durch die Anwendung: apply inc erhalten wir:

$$\begin{aligned} \alpha &= \mathbf{int} \\ \beta &= \mathbf{int} \\ \alpha[\mathsf{apply}] &= (\mathbf{int} \to \mathbf{int}) \to \mathbf{int} \to \mathbf{int} \end{aligned}$$

$\Longrightarrow$ Typ-Fehler ???

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

... im Beispiel:

```
let apply  =  fn f ⇒ fn x ⇒ f x;
    inc    =  fn y ⇒ y + 1;
    single =  fn y ⇒ y : [ ]
in  (fn f ⇒ fn x ⇒ f x) single
   ((fn f ⇒ fn x ⇒ f x) inc 1)
end
```

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

... im Beispiel:

```
let apply  =  fn f ⇒ fn x ⇒ f x;
    inc    =  fn y ⇒ y + 1;
    single =  fn y ⇒ y : [ ]
in  (fn f ⇒ fn x ⇒ f x) single
   ((fn f ⇒ fn x ⇒ f x) inc 1)
end
```

\+ Die beiden Teilausdrücke (**fn** f ⇒ **fn** x ⇒ $f\,x$) 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 :-((

Idee 2: Kopiere die Typen für jede Benutzung ...

- Wir erweitern Typen zu Typ-Schemata:

$$
\begin{array}{lcl}
t & ::= & \alpha \mid \textbf{bool} \mid \textbf{int} \mid (t_1, \ldots, t_m) \mid \mathsf{list}\ t \mid t_1 \to t_2 \\
\sigma & ::= & t \mid \forall\, \alpha_1, \ldots, \alpha_k . t
\end{array}
$$

- Achtung: Der Operator $\forall$ erscheint nur auf dem Top-Level !!!
- Typ-Schemata werden für **let**-definierte Variablen eingeführt.
- Bei deren Benutzung wird der Typ im Schema mit frischen Typ-Variablen instantiiert ...

Neue Regeln:

$$\text{Inst:}\quad \frac{\Gamma(x) = \forall\, \alpha_1, \ldots, \alpha_k.t}{\Gamma \;\vdash\; x \;:\; t[t_1/\alpha_1, \ldots, t_k/\alpha_k]} \qquad (t_1, \ldots, t_k \quad \text{beliebig})$$

$$\text{Let:}\quad \frac{\begin{array}{llll} \Gamma_0 & \vdash & e_1 \;:\; t_1 & \Gamma_1 = \Gamma_0 \oplus \{x_1 \mapsto \mathsf{close}\, t_1\, \Gamma_0\} \\ & \ldots & & \ldots \\ \Gamma_{m-1} & \vdash & e_m \;:\; t_m & \Gamma_m = \Gamma_{m-1} \oplus \{x_m \mapsto \mathsf{close}\, t_m\, \Gamma_{m-1}\} \\ \Gamma_m & \vdash & e_0 \;:\; t_0 & \end{array}}{\Gamma_0 \;\vdash\; (\textbf{let}\; x_1 = e_1; \ldots; x_m = e_m\; \textbf{in}\; e_0) \;:\; t_0}$$

Der Aufruf close $t\ \Gamma$ macht alle Typ-Variablen in t generisch (d.h. instantiierbar), die nicht auch in Γ vorkommen ...

$$\begin{array}{l}\textbf{fun}\ \mathsf{close}\ t\ \Gamma\ =\ \textbf{let}\\ \qquad\textbf{val}\ \alpha_1,\ldots,\alpha_k = \mathsf{free}\,(t) \setminus \mathsf{free}\,(\Gamma)\\ \qquad\textbf{in}\ \forall\,\alpha_1,\ldots,\alpha_k.\,t\\ \qquad\textbf{end}\end{array}$$

Eine Instantiierung mit frischen Typ-Variablen leistet die Funktion:

$$\begin{array}{l}\textbf{fun}\ \mathsf{inst}\ \sigma\ =\ \textbf{let}\\ \qquad\textbf{val}\ \forall\,\alpha_1,\ldots,\alpha_k.\,t = \sigma\\ \qquad\textbf{val}\ \beta_1 = \mathsf{new}()\quad\ldots\quad\textbf{val}\ \beta_k = \mathsf{new}()\\ \qquad\textbf{in}\ t[\beta_1/\alpha_1,\ldots,\beta_k/\alpha_k]\\ \qquad\textbf{end}\end{array}$$

Der Algorithmus $\mathcal{W}$ (erweitert):

$$
\begin{array}{lll}
\ldots & & \\
\mid\ x & \rightarrow & (\mathsf{inst}\ (\Gamma(x)), \theta) \\
\mid\ (\textbf{let}\ x_1 = e_1; \ldots; x_m = e_m\ \textbf{in}\ e_0) & & \\
 & \rightarrow & \textbf{let}\ \textbf{val}\ (t_1, \theta) = \mathcal{W}\ e_1\ (\Gamma, \theta) \\
 & & \quad \textbf{val}\ \sigma_1 = \mathsf{close}\ (\theta\ t_1)\ (\theta\ \Gamma) \\
 & & \quad \textbf{val}\ \Gamma = \Gamma \oplus \{x_1 \mapsto \sigma_1\} \\
 & & \quad \ldots \\
 & & \quad \textbf{val}\ (t_m, \theta) = \mathcal{W}\ e_m\ (\Gamma, \theta) \\
 & & \quad \textbf{val}\ \sigma_m = \mathsf{close}\ (\theta\ t_m)\ (\theta\ \Gamma) \\
 & & \quad \textbf{val}\ \Gamma = \Gamma \oplus \{x_m \mapsto \sigma_m\} \\
 & & \quad \textbf{val}\ (t_0, \theta) = \mathcal{W}\ e_0\ (\Gamma, \theta) \\
 & & \textbf{in}\ (t_0, \theta) \\
 & & \textbf{end}
\end{array}
$$

Beispiel:

$$
\begin{array}{lll}
\textbf{let}\ \mathsf{apply} & = & \textbf{fn}\ f \Rightarrow \textbf{fn}\ x \Rightarrow f\ x; \\
\quad \mathsf{inc} & = & \textbf{fn}\ y \Rightarrow y + 1; \\
\quad \mathsf{single} & = & \textbf{fn}\ y \Rightarrow y : [\,] \\
\textbf{in}\ \mathsf{apply}\ \mathsf{single}\ (\mathsf{apply}\ \mathsf{inc}\ 1) & & \\
\textbf{end} & &
\end{array}
$$

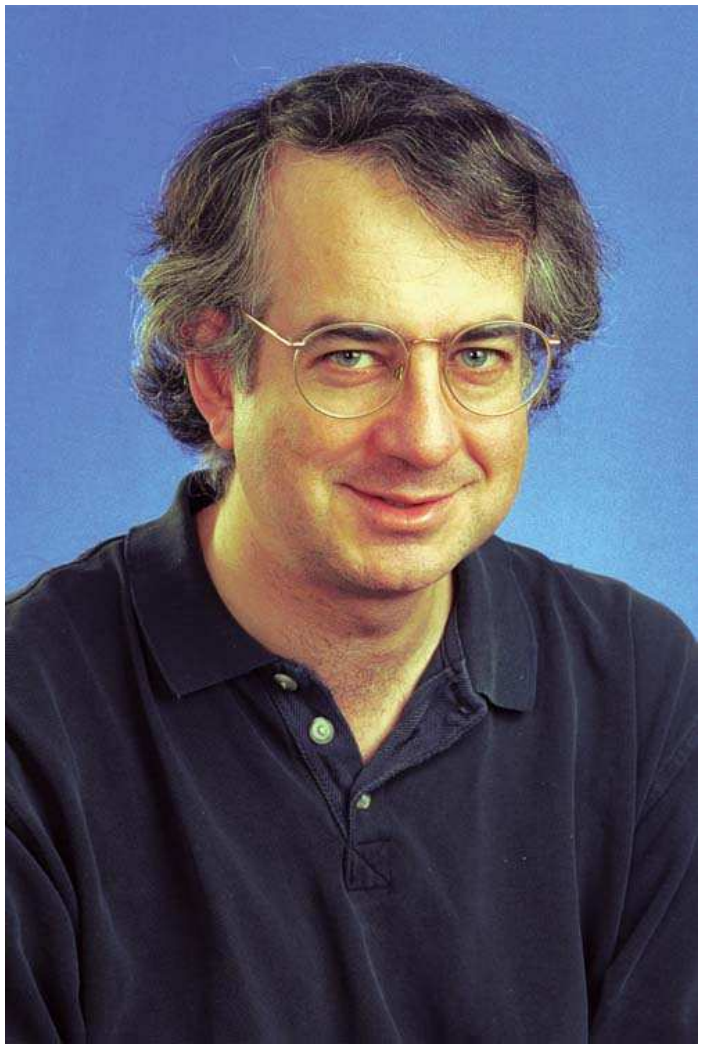
Wir finden:

$$
\begin{array}{lll}
\alpha[\mathsf{apply}] & = & \forall\, \alpha, \beta.\, (\alpha \to \beta) \to \alpha \to \beta \\
\alpha[\mathsf{inc}] & = & \textbf{int} \to \textbf{int} \\
\alpha[\mathsf{single}] & = & \forall\, \gamma.\, \gamma \to \mathsf{list}\ \gamma
\end{array}
$$

Bemerkungen:

- Der erweiterte Algorithmus berechnet nach wie vor allgemeinste Typen :-)
- Instantiierung von Typ-Schemata bei jeder Benutzung ermöglicht polymorphe Funktionen sowie modulare Typ-Inferenz :-))
- Die Möglichkeit der Instantiierung erlaubt die Codierung von DEXPTIME-schwierigen Problemen in die Typ-Inferenz ??

  ... ein in der Praxis eher marginales Problem :-)
- Die Einführung von Typ-Schemata ist nur für nicht-rekursive Definitionen möglich: die Ermittlung eines allgemeinsten Typ-Schemas für rekursive Definitionen ist nicht berechenbar !!!



Harry Mairson, Brandeis University

Seiteneffekte

- Für ein elegantes Programmieren sind gelegentlich Variablen, deren Wert geändert werden kann, ganz nützlich :-)
- Darum erweitern wir unsere kleine Programmiersprache um Referenzen:

$$e \;::=\; \ldots \;\mid\; \textbf{ref}\; e \;\mid\; !e \;\mid\; e_1 := e_2$$

Seiteneffekte

- Für ein elegantes Programmieren sind gelegentlich Variablen, deren Wert geändert werden kann, ganz nützlich :-)
- Darum erweitern wir unsere kleine Programmiersprache um Referenzen:

$$e \;::=\; \ldots \;\mid\; \textbf{ref}\; e \;\mid\; !e \;\mid\; e_1 := e_2$$

Beispiel:

```
let count = ref 0;
    new   = fn () ⇒ let
                      ret = !count;
                      _   = count := ret + 1
                    in ret
in new() + new()
```