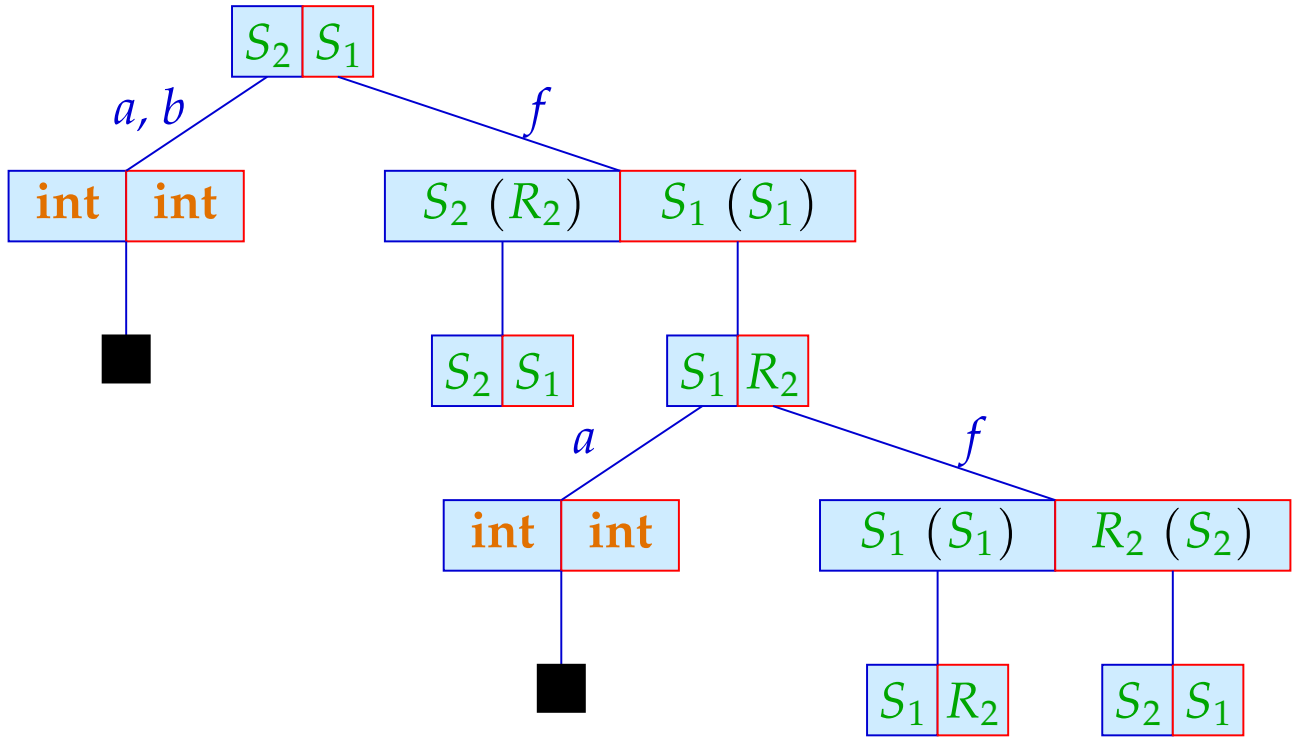
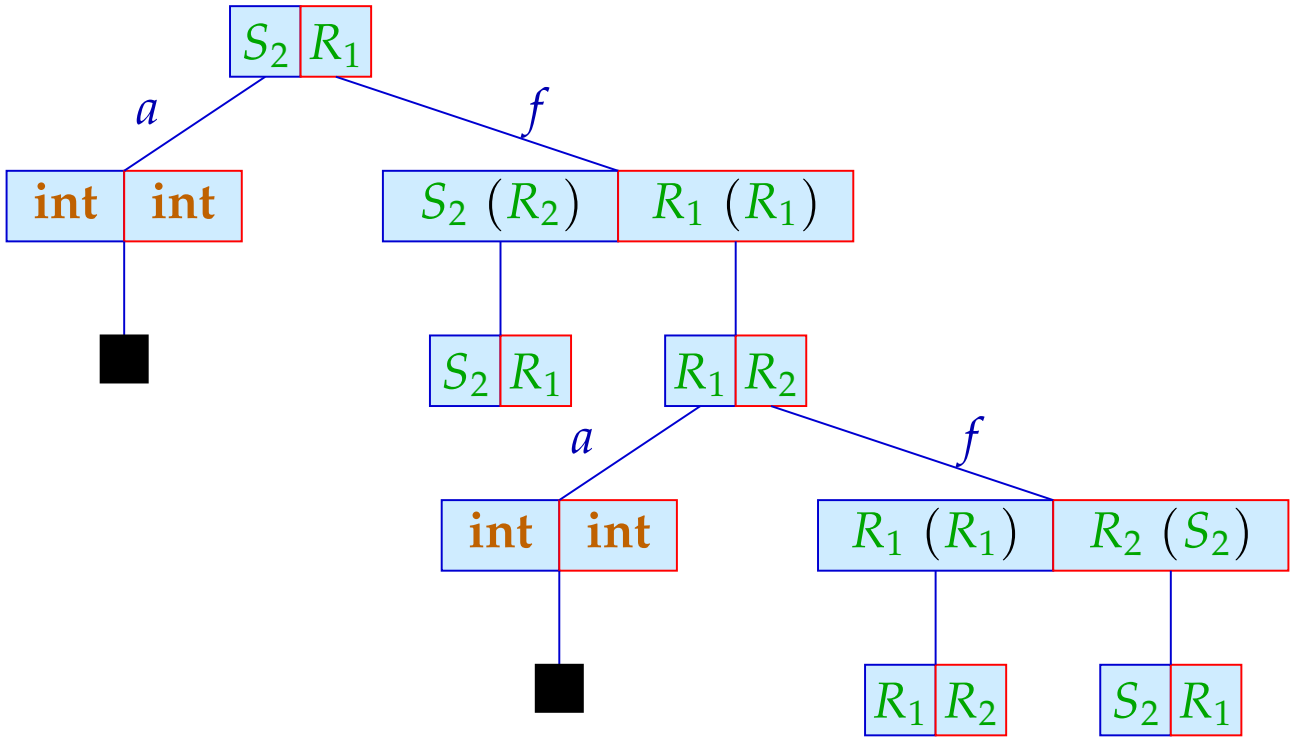

S_2 S_1
a, b
f
int int
S_2 (R_2)
S_1 (S_1)
S_2 S_1
S_1 R_2
a
f
int int
S_1 (S_1)
R_2 (S_2)
S_1 R_2
S_2 S_1


S_2 R_1
a
f
int int
S_2 (R_2)
R_1 (R_1)
S_2 R_1
R_1 R_2
a
f
int int
R_1 (R_1)
R_2 (S_2)
R_1 R_2
S_2 R_1

Diskussion:

- Um die Beweisbäume nicht in den Himmel wachsen zu lassen, wurden einige Zwischenknoten ausgelassen :-)
- Strukturelle Teiltypen sind sehr mächtig und deshalb nicht ganz leicht zu durchschauen.
- Java verallgemeinert Strukturen zu Objekten / Klassen.
- Teiltyp-Beziehungen zwischen Klassen müssen explizit deklariert werden :-)
- Durch Vererbung wird sichergestellt, dass Unterklassen über die (sichtbaren) Komponenten der Oberklasse verfügen :-))
- Überdecken einer Komponente mit einer anderen gleichen Namens ist möglich — aber nur, wenn diese keine Methode ist :-(

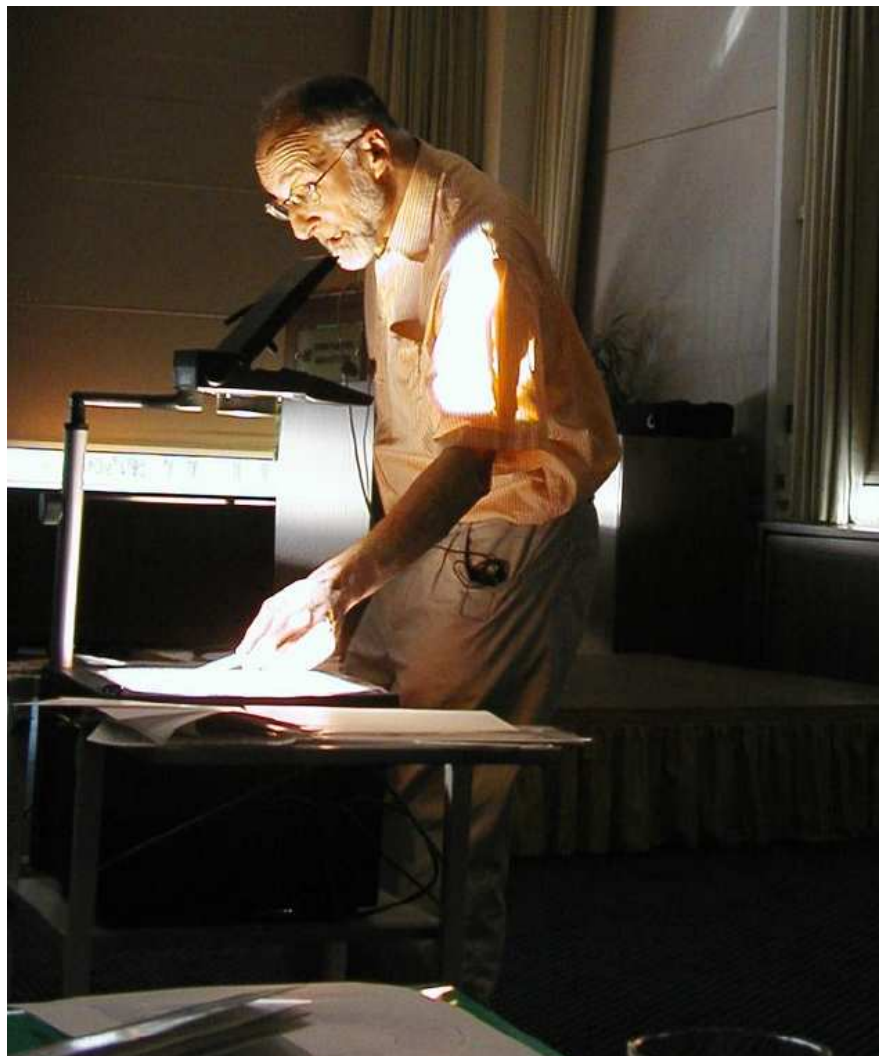
3.3 Inferieren von Typen

- Im Gegensatz zu imperativen Sprachen kann in funktionalen Programmiersprachen der Typ von Bezeichnern (i.a.) weggelassen werden.
- Diese werden dann automatisch hergeleitet :-)

Beispiel:

$$\begin{array}{lcl} \textbf{fun}\ \ \mathsf{fac}\ \mathsf{x} & = & \textbf{if}\ \ \mathsf{x} \leq 0\ \ \textbf{then}\ \ 1 \\ & & \textbf{else}\ \ \mathsf{x} \cdot \mathsf{fac}\ (\mathsf{x} - 1) \end{array}$$

Dafür findet der SML-Compiler: $\mathsf{fac}\ :\ \ \textbf{int} \rightarrow \textbf{int}$



Robin (Dumbledore) Milner, Edinburgh

Idee: J.R. Hindley, R. Milner

Stelle Axiome und Regeln auf, die den Typ eines Ausdrucks in Beziehung setzen zu den Typen seiner Teilausdrücke :-)

Der Einfachkeit halber betrachten wir nur eine funktionale Kernsprache ...

$$
\begin{array}{rcl}
e & ::= & b \mid x \mid (\Box_1\ e) \mid (e_1\ \Box_2\ e_2) \\
 & \mid & (\textbf{if}\ e_0\ \textbf{then}\ e_1\ \textbf{else}\ e_2) \\
 & \mid & (e_1, \ldots, e_k) \mid [\,] \mid (e_1\ :\ e_2) \\
 & \mid & (\textbf{case}\ e_0\ \textbf{of}\ [\,] \rightarrow e_1;\ h : t \rightarrow e_2) \\
 & \mid & (e_1\ e_2) \mid (\textbf{fn}\ (x_1, \ldots, x_m) \Rightarrow e) \\
 & \mid & (\textbf{letrec}\ x_1 = e_1; \ldots; x_n = e_n\ \textbf{in}\ e_0) \\
 & \mid & (\textbf{let}\ x_1 = e_1; \ldots; x_n = e_n\ \textbf{in}\ e_0)
\end{array}
$$

Beispiel:

$$
\begin{array}{lll}
\textbf{letrec} & \mathsf{rev} & = \quad \textbf{fn}\ x \Rightarrow \mathsf{r}\ x\ [\,]; \\
 & \mathsf{r} & = \quad \textbf{fn}\ x \Rightarrow \textbf{fn}\ y \Rightarrow\ \textbf{case}\ x\ \textbf{of} \\
 & & \qquad\qquad [\,] \rightarrow y; \\
 & & \qquad\qquad h : t \rightarrow \mathsf{r}\ t\ (h : y) \\
\textbf{in} & \mathsf{rev}\ (1 : 2 : 3 : [\,]) &
\end{array}
$$

Wir benutzen die üblichen Präzedenz-Regeln und Assoziativitäten, um hässliche Klammern zu sparen :-)

Als einzige Datenstrukturen betrachten wir Tupel und List :-))

Wir benutzen eine Syntax von Typen, die an SML angelehnt ist ...

$$t \quad ::= \quad \textbf{int} \mid \textbf{bool} \mid (t_1, \ldots, t_m) \mid \mathsf{list}\ t \mid t_1 \rightarrow t_2$$

Wir betrachten wieder Typ-Aussagen der Form:

$$\Gamma \vdash e : t$$

Wir benutzen eine Syntax von Typen, die an SML angelehnt ist ...

$$t \quad ::= \quad \textbf{int} \mid \textbf{bool} \mid (t_1, \ldots, t_m) \mid \mathsf{list}\ t \mid t_1 \rightarrow t_2$$

Wir betrachten wieder Typ-Aussagen der Form:

$$\Gamma \vdash e : t$$

Axiome:

Const:	$\Gamma \vdash c : t_c$	(t_c Typ der Konstante c)
Nil:	$\Gamma \vdash [\,] : \mathsf{list}\ t$	(t beliebig)
Var:	$\Gamma \vdash x : \Gamma(x)$	(x Variable)

Regeln:

Op:
$$\frac{\Gamma \vdash e_1 : \textbf{int} \qquad \Gamma \vdash e_2 : \textbf{int}}{\Gamma \vdash e_1 + e_2 : \textbf{int}}$$

If:
$$\frac{\Gamma \vdash e_0 : \textbf{bool} \qquad \Gamma \vdash e_1 : t \qquad \Gamma \vdash e_2 : t}{\Gamma \vdash (\textbf{if } e_0 \textbf{ then } e_1 \textbf{ else } e_2) : t}$$

Tupel:
$$\frac{\Gamma \vdash e_1 : t_1 \quad \ldots \quad \Gamma \vdash e_m : t_m}{\Gamma \vdash (e_1, \ldots, e_m) : (t_1, \ldots, t_m)}$$

App:
$$\frac{\Gamma \vdash e_1 : t_1 \to t_2 \qquad \Gamma \vdash e_2 : t_1}{\Gamma \vdash (e_1\ e_2) : t_2}$$

Fun:
$$\frac{\Gamma \oplus \{x_1 \mapsto t_1, \ldots, x_m \mapsto t_m\} \vdash e : t}{\Gamma \vdash \textbf{fn } (x_1, \ldots, x_m) \Rightarrow e : (t_1, \ldots, t_m) \to t}$$

...

$\ldots$

Cons: $$\frac{\Gamma \vdash e_1 : t \qquad \Gamma \vdash e_2 : \text{list } t}{\Gamma \vdash (e_1 : e_2) : \text{list } t}$$

Case: $$\frac{\Gamma \vdash e_0 : \text{list } t_1 \qquad \Gamma \vdash e_1 : t \qquad \Gamma \oplus \{x \mapsto t_1, y \mapsto \text{list } t_1\} \vdash e_2 : t}{\Gamma \vdash (\textbf{case } e_0 \textbf{ of } [\,] \to e_1;\ x : y \to e_2) : t}$$

Letrec: $$\frac{\Gamma' \vdash e_1 : t_1 \quad \ldots \quad \Gamma' \vdash e_m : t_m \quad \Gamma' \vdash e_0 : t}{\Gamma \vdash (\textbf{letrec } x_1 = e_1; \ldots; x_m = e_m \textbf{ in } e_0) : t}$$

wobei $\Gamma' = \Gamma \oplus \{x_1 \mapsto t_1, \ldots, x_m \mapsto t_m\}$

$\ldots$

Cons: $$\frac{\Gamma \vdash e_1 : t \qquad \Gamma \vdash e_2 : \text{list } t}{\Gamma \vdash (e_1 : e_2) : \text{list } t}$$

Case: $$\frac{\Gamma \vdash e_0 : \text{list } t_1 \qquad \Gamma \vdash e_1 : t \qquad \Gamma \oplus \{x \mapsto t_1, y \mapsto \text{list } t_1\} \vdash e_2 : t}{\Gamma \vdash (\textbf{case } e_0 \textbf{ of } [\,] \to e_1;\ x : y \to e_2) : t}$$

Letrec: $$\frac{\Gamma' \vdash e_1 : t_1 \quad \ldots \quad \Gamma' \vdash e_m : t_m \quad \Gamma' \vdash e_0 : t}{\Gamma \vdash (\textbf{letrec } x_1 = e_1; \ldots; x_m = e_m \textbf{ in } e_0) : t}$$

wobei $\Gamma' = \Gamma \oplus \{x_1 \mapsto t_1, \ldots, x_m \mapsto t_m\}$

Könnten wir die Typen für alle Variablen-Vorkommen raten, ließe sich mithilfe der Regeln überprüfen, dass unsere Wahl korrekt war :-)

$\ldots$

Cons: $$\frac{\Gamma \vdash e_1 : t \qquad \Gamma \vdash e_2 : \text{list } t}{\Gamma \vdash (e_1 : e_2) : \text{list } t}$$

Case: $$\frac{\Gamma \vdash e_0 : \text{list } t_1 \qquad \Gamma \vdash e_1 : t \qquad \Gamma \oplus \{x \mapsto t_1, y \mapsto \text{list } t_1\} \vdash e_2 : t}{\Gamma \vdash (\textbf{case } e_0 \textbf{ of } [\,] \to e_1;\ x : y \to e_2) : t}$$

Letrec: $$\frac{\Gamma' \vdash e_1 : t_1 \quad \ldots \quad \Gamma' \vdash e_m : t_m \quad \Gamma' \vdash e_0 : t}{\Gamma \vdash (\textbf{letrec } x_1 = e_1; \ldots; x_m = e_m \textbf{ in } e_0) : t}$$

wobei $\Gamma' = \Gamma \oplus \{x_1 \mapsto t_1, \ldots, x_m \mapsto t_m\}$

Könnten wir die Typen für alle Variablen-Vorkommen raten, ließe sich mithilfe der Regeln überprüfen, dass unsere Wahl korrekt war :-)

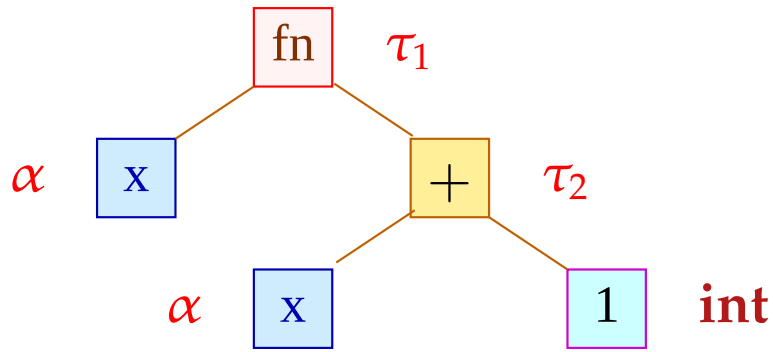
Wie raten wir die Typen der Variablen ???

Idee:

- Mache die Namen der verschiedenen Variablen eindeutig.
- Führe Typ-Variablen für die unbekannten Typen der Variablen und Teilausdrücke ein.
- Sammle die Gleichungen, die notwendigerweise zwischen den Typ-Variablen gelten müssen.
- Finde für diese Gleichungen Lösungen :-)

Beispiel:

$$\textbf{fn}\ x \Rightarrow x + 1$$



Gleichungen:

$$\begin{aligned} \tau_1 &= \alpha \rightarrow \tau_2 \\ \tau_2 &= \mathbf{int} \\ \alpha &= \mathbf{int} \end{aligned}$$

Wir schließen: $\tau_1 = \mathbf{int} \rightarrow \mathbf{int}$

Für jede Programm-Variable x und für jedes Vorkommen eines Teilausdrucks e führen wir die Typ-Variable $\alpha[x]$ bzw. $\tau[e]$ ein.

Jede Regel-Anwendung gibt dann Anlass zu einigen Gleichungen ...

Const:	$e \equiv c$	$\Longrightarrow$	$\tau[e] = \tau_c$
Nil:	$e \equiv [\,]$	$\Longrightarrow$	$\tau[e] = \mathsf{list}\,\alpha \qquad (\alpha \quad \text{neu})$
Var:	$e \equiv x$	$\Longrightarrow$	$\tau[e] = \alpha[x]$
Op:	$e \equiv e_1 + e_2$	$\Longrightarrow$	$\tau[e] = \tau[e_1] = \tau[e_2] = \mathbf{int}$
Tupel:	$e \equiv (e_1, \ldots, e_m)$	$\Longrightarrow$	$\tau[e] = (\tau[e_1], \ldots, \tau[e_m])$
Cons:	$e \equiv e_1 : e_2$	$\Longrightarrow$	$\tau[e] = \tau[e_2] = \mathsf{list}\ \tau[e_1]$
	$\ldots$		

$\ldots$

If:	$e \equiv \textbf{if}\ e_0\ \textbf{then}\ e_1\ \textbf{else}\ e_2$	$\Longrightarrow$	$\tau[e_0] = \textbf{bool}$ $\tau[e] = \tau[e_1] = \tau[e_2]$
Case:	$e \equiv \textbf{case}\ e_0\ \textbf{of}\ [\,] \rightarrow e_1;\ x : y \rightarrow e_2$	$\Longrightarrow$	$\tau[e_0] = \alpha[y] = \mathsf{list}\ \alpha[x]$ $\tau[e] = \tau[e_1] = \tau[e_2]$
Fun:	$e \equiv \textbf{fn}\ (x_1, \ldots, x_m) \Rightarrow e_1$	$\Longrightarrow$	$\tau[e] = (\alpha[x_1], \ldots, \alpha[x_m]) \rightarrow \tau[e_1]$
App:	$e \equiv e_1\ e_2$	$\Longrightarrow$	$\tau[e_1] = \tau[e_2] \rightarrow \tau[e]$
Letrec:	$e \equiv \textbf{letrec}\ x_1 = e_1; \ldots; x_m = e_m\ \textbf{in}\ e_0$	$\Longrightarrow$	$\alpha[x_1] = \tau[e_1] \ldots$ $\alpha[x_m] = \tau[e_m]$ $\tau[e] = \tau[e_0]$

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 :-)

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:

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

```
...
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
```

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

Diskussion:

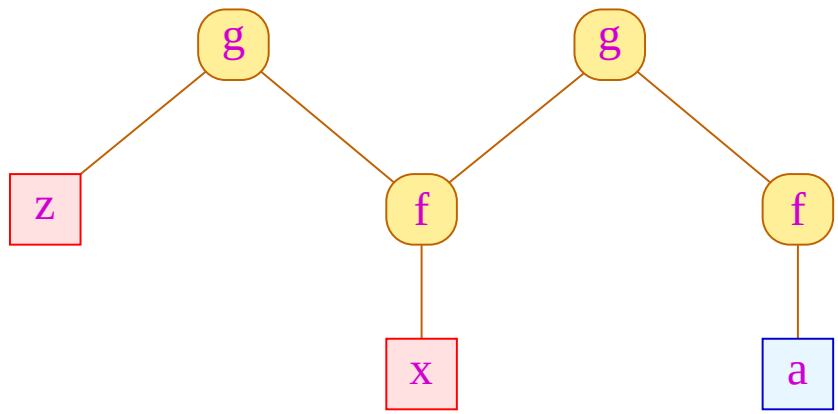
- Der Algorithmus startet mit $\mathsf{unifyList}\ [(s_1,t_1),\ldots,(s_m,t_m)]\ \{\ \}$ ...
- Der Algorithmus liefert sogar eine idempotente allgemeinste Lösung :-)
- Leider hat er möglicherweise exponentielle Laufzeit :-(
- Lässt sich das verbessern ???

Idee:

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

... im Beispiel:

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

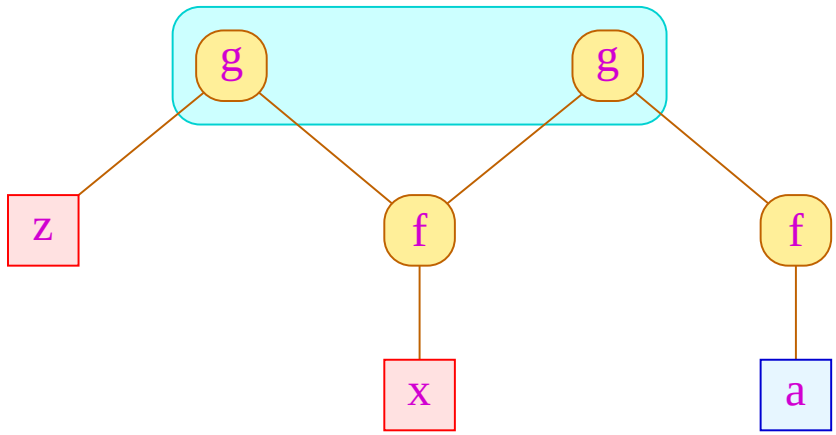


Idee:

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

... im Beispiel:

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

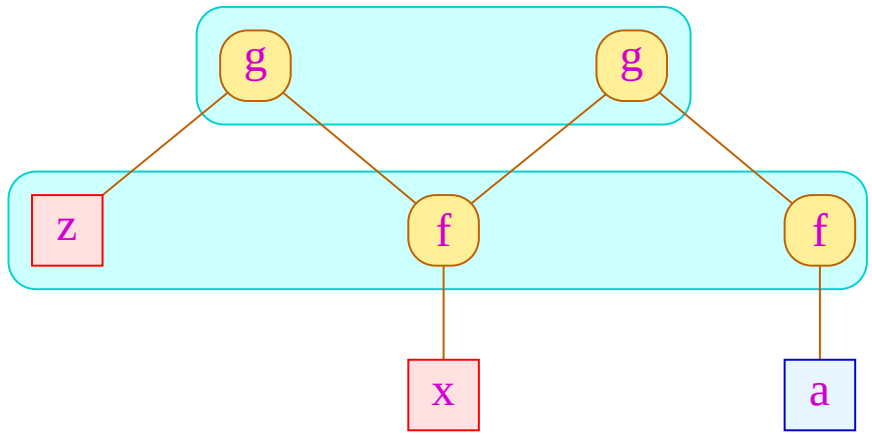


Idee:

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

... im Beispiel:

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

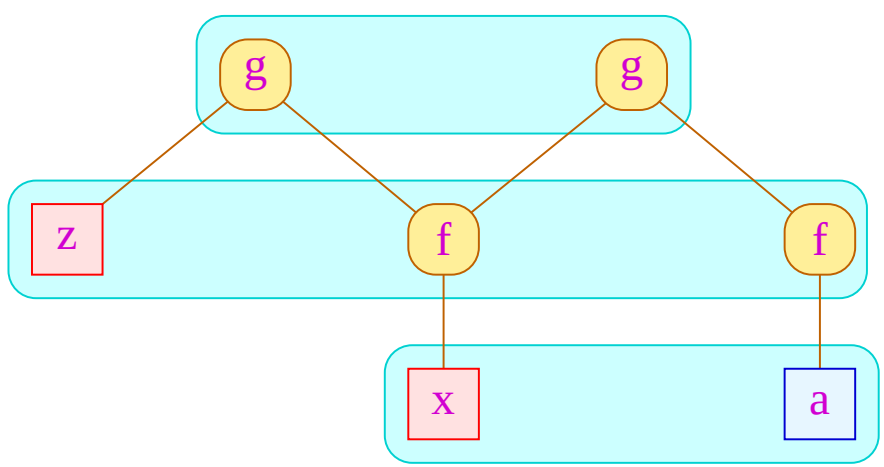


Idee:

- 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:

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

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

  $\rightarrow$ 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 $\pi_0 = \{\{v\} \mid v \in \mathsf{Nodes}\}$

  $\rightarrow$ 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 ...

```
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(v);
                W = (u_1, v_1) :: ... :: (u_k, v_k) :: W;
            }
    }
}
```

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 :-)

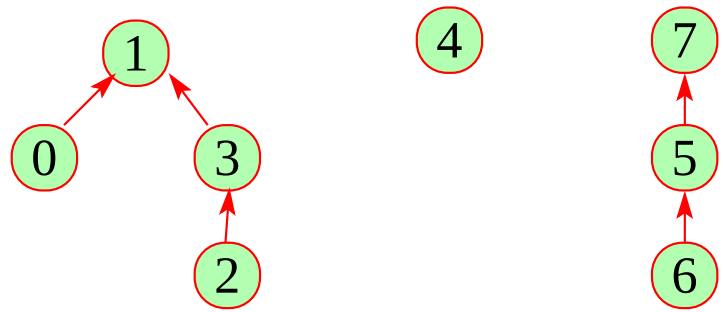
Idee:

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 ...

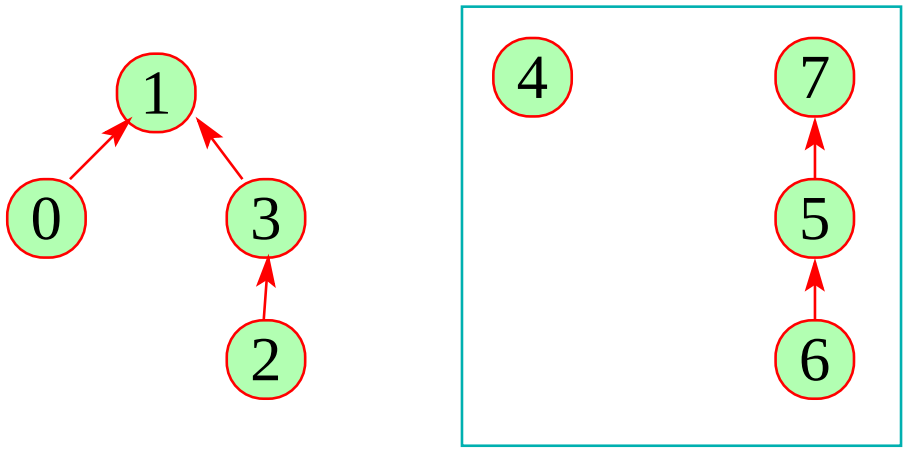


0	1	2	3	4	5	6	7

1	1	3	1	4	7	5	7

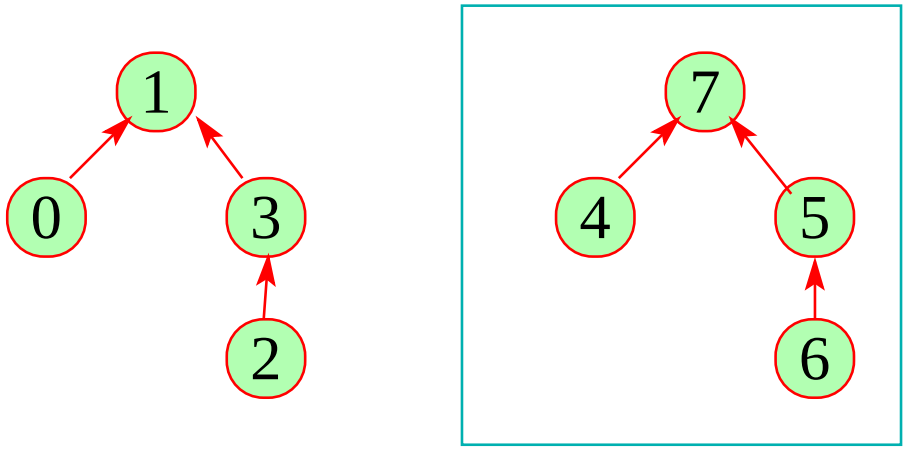
→ find (π, u) folgt den Vater-Verweisen :-)

→ 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

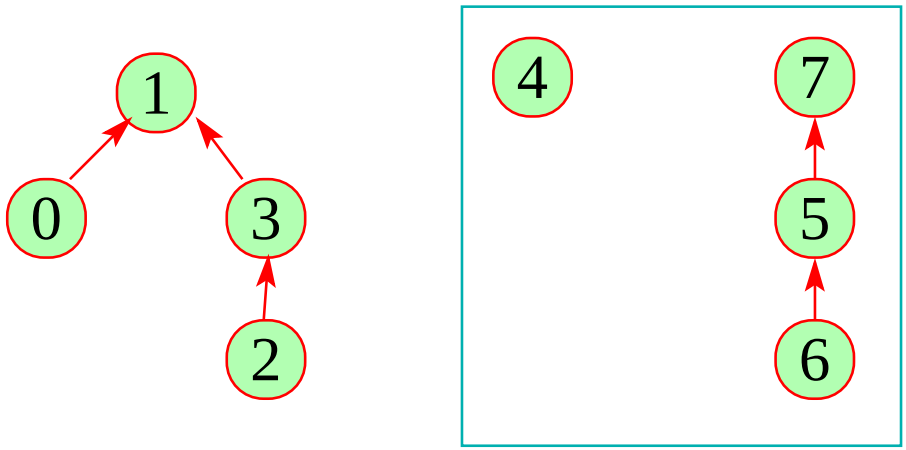
1	1	3	1	7	7	5	7

Die Kosten:

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

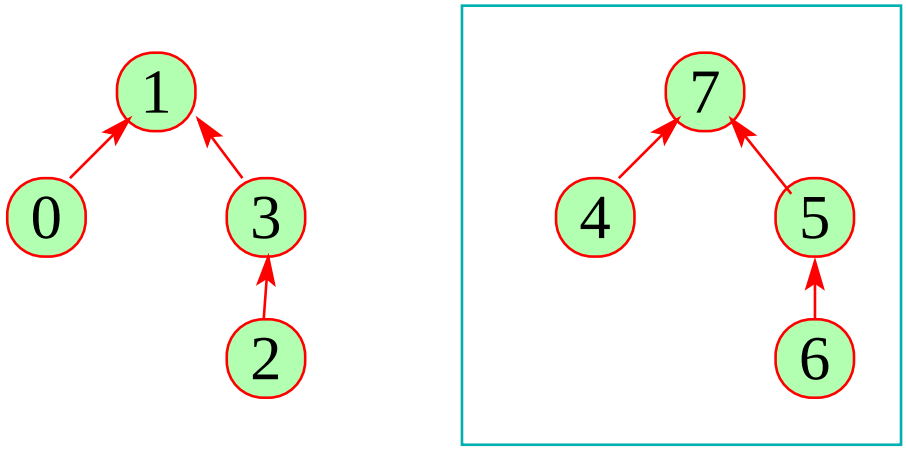
Strategie zur Vermeidung tiefer Bäume:

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



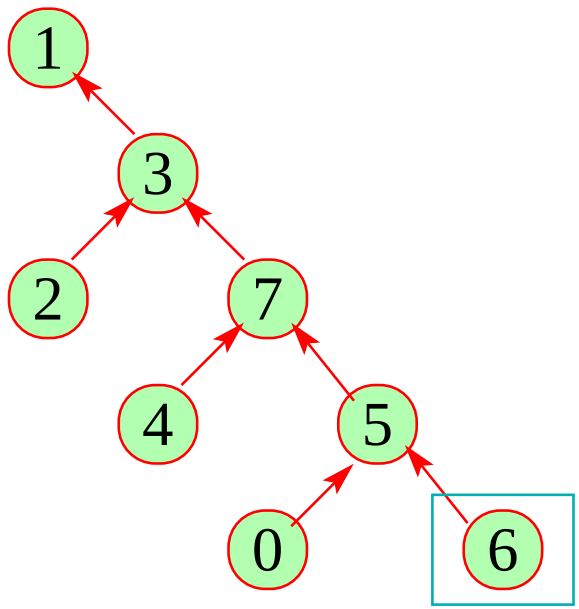
0	1	2	3	4	5	6	7

1	1	3	1	4	7	5	7

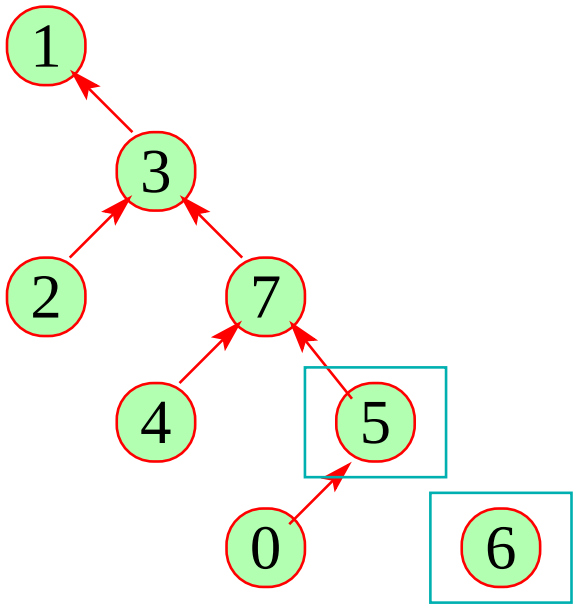


0	1	2	3	4	5	6	7

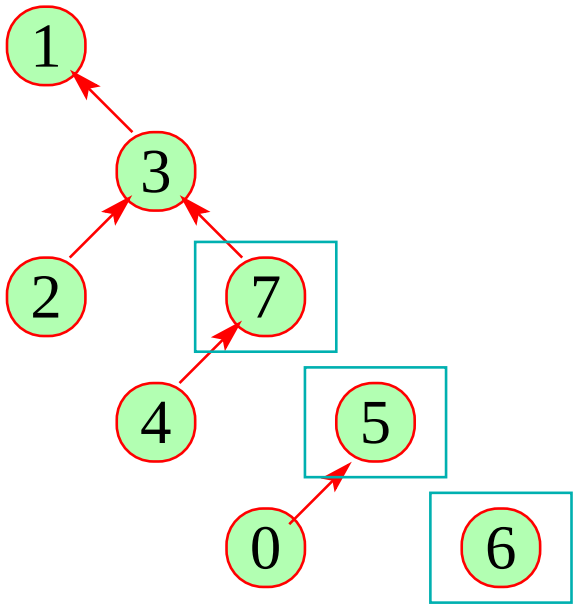
1	1	3	1	7	7	5	7



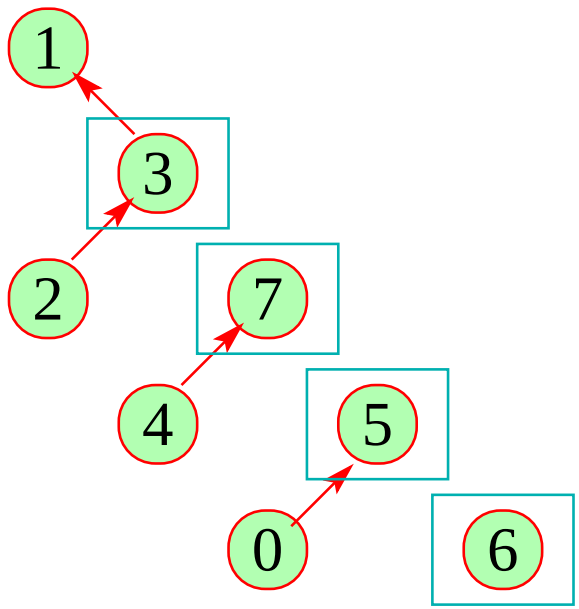
0	1	2	3	4	5	6	7
5	1	3	1	7	7	5	3



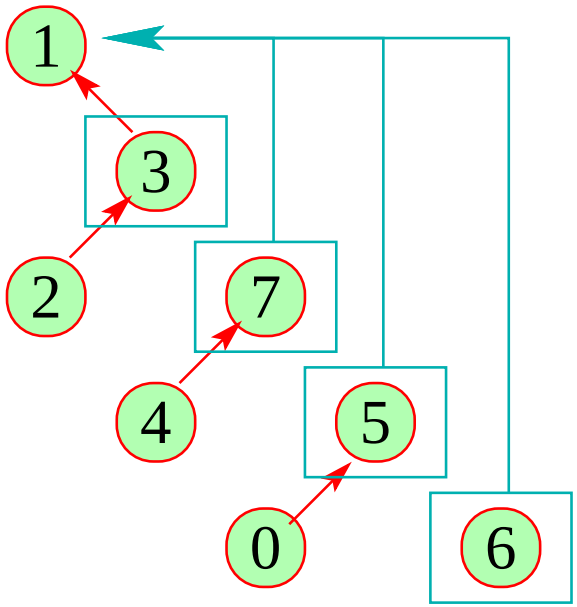
0	1	2	3	4	5	6	7
5	1	3	1	7	7	5	3



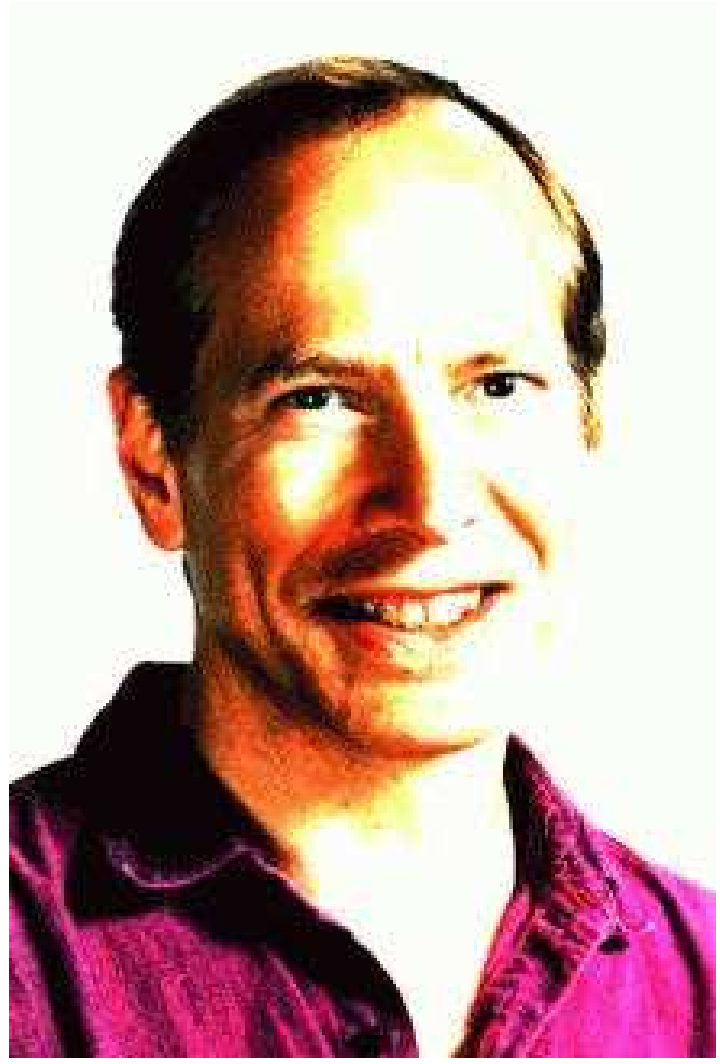
0	1	2	3	4	5	6	7
5	1	3	1	7	7	5	3



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:

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

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.):

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

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

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

$\ldots$

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

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

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

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

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

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

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

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

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 :-}