Teiltypen

- Auf den arithmetischen Basistypen **char**, **int**,**long**, ... gibt es i.a. eine reichhaltige Teiltypen-Beziehungen.
- Dabei bedeutet $t_1 \leq t_2$, dass die Menge der Werte vom Typ t_1

  (1) eine Teilmenge der Werte vom Typ t_2 sind :-)

  (2) in einen Wert vom Typ t_2 konvertiert werden können :-)

  (3) die Anforderungen an Werte vom Typ t_2 erfüllen ...

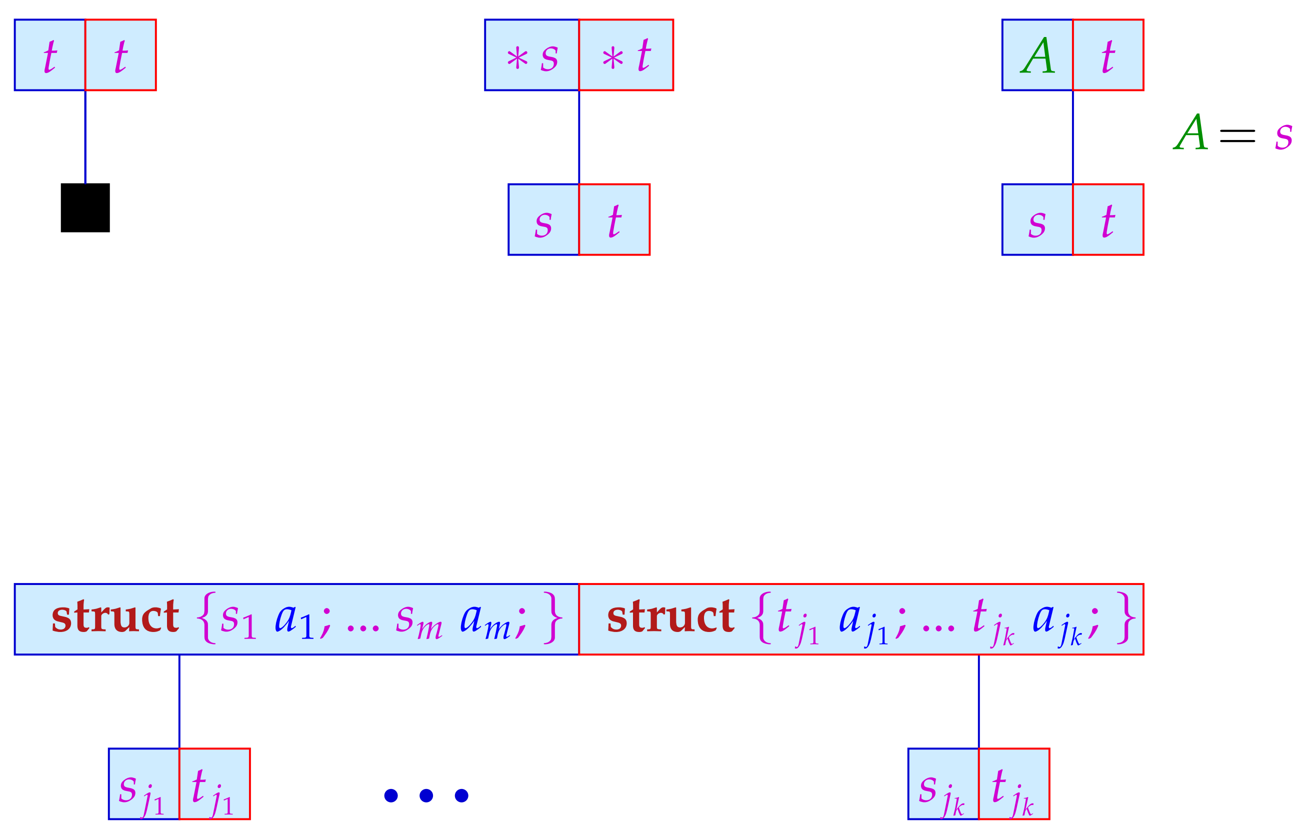
$\Longrightarrow$

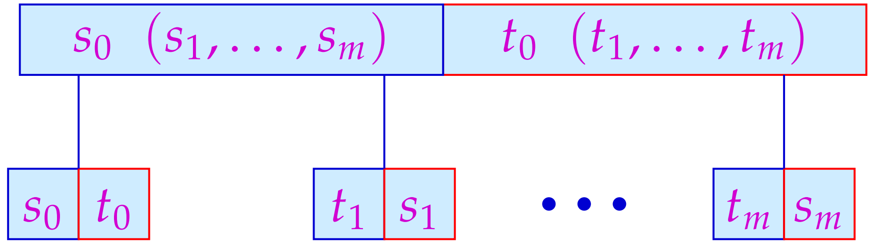
Erweitere Teiltypen-Beziehungen der Basistypen auf komplexe Typen :-)

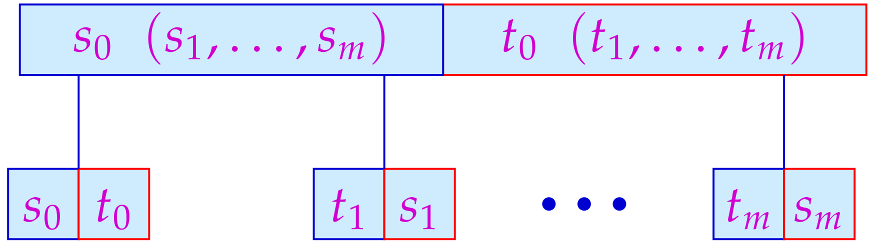
Beispiel:

```
string extractInfo (struct { string info; }  x) {
        return x.info;
    }
```

- Offenkundig funktioniert extractInfo für alle Argument-Strukturen, die eine Komponente **string** info besitzen :-)
- Die Idee ist vergleichbar zur Anwendbarkeit auf Unterklassen (aber allgemeiner :-)
- Wann $t_1 \leq t_2$ gelten soll, beschreiben wir durch Regeln ...


t t
$*s$ $*t$
s t
A t
s t
$A = s$
struct $\{s_1\ a_1;\ \dots\ s_m\ a_m;\ \}$ **struct** $\{t_{j_1}\ a_{j_1};\ \dots\ t_{j_k}\ a_{j_k};\ \}$
s_{j_1} t_{j_1}
s_{j_k} t_{j_k}

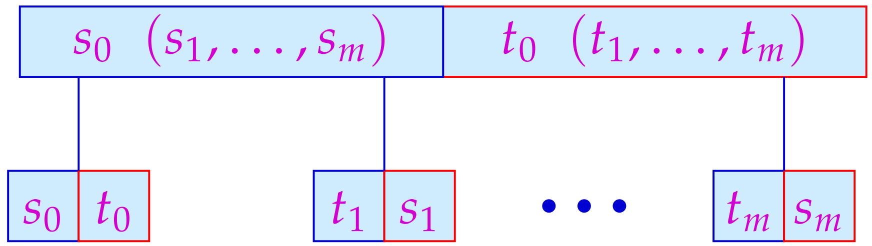

$s_0 \ (s_1, \ldots, s_m)$
$t_0 \ (t_1, \ldots, t_m)$
s_0
t_0
t_1
s_1
t_m
s_m



Beispiele:

$$\textbf{struct}\ \{\textbf{int}\ a;\ \textbf{int}\ b;\} \quad \leq \quad \textbf{struct}\ \{\textbf{float}\ a;\}$$

$$\textbf{int}\ (\textbf{int}) \quad \not\leq \quad \textbf{float}\ (\textbf{float})$$



Beispiele:

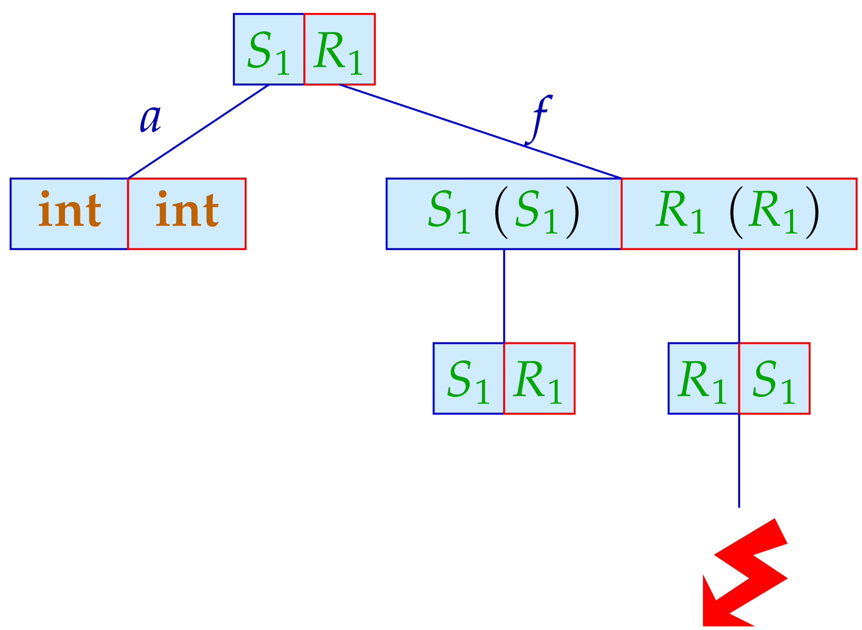
$$\textbf{struct}\ \{\textbf{int}\ a;\ \ \textbf{int}\ b;\} \quad \leq \quad \textbf{struct}\ \{\textbf{float}\ a;\}$$
$$\textbf{int}\ (\textbf{int}) \quad \not\leq \quad \textbf{float}\ (\textbf{float})$$

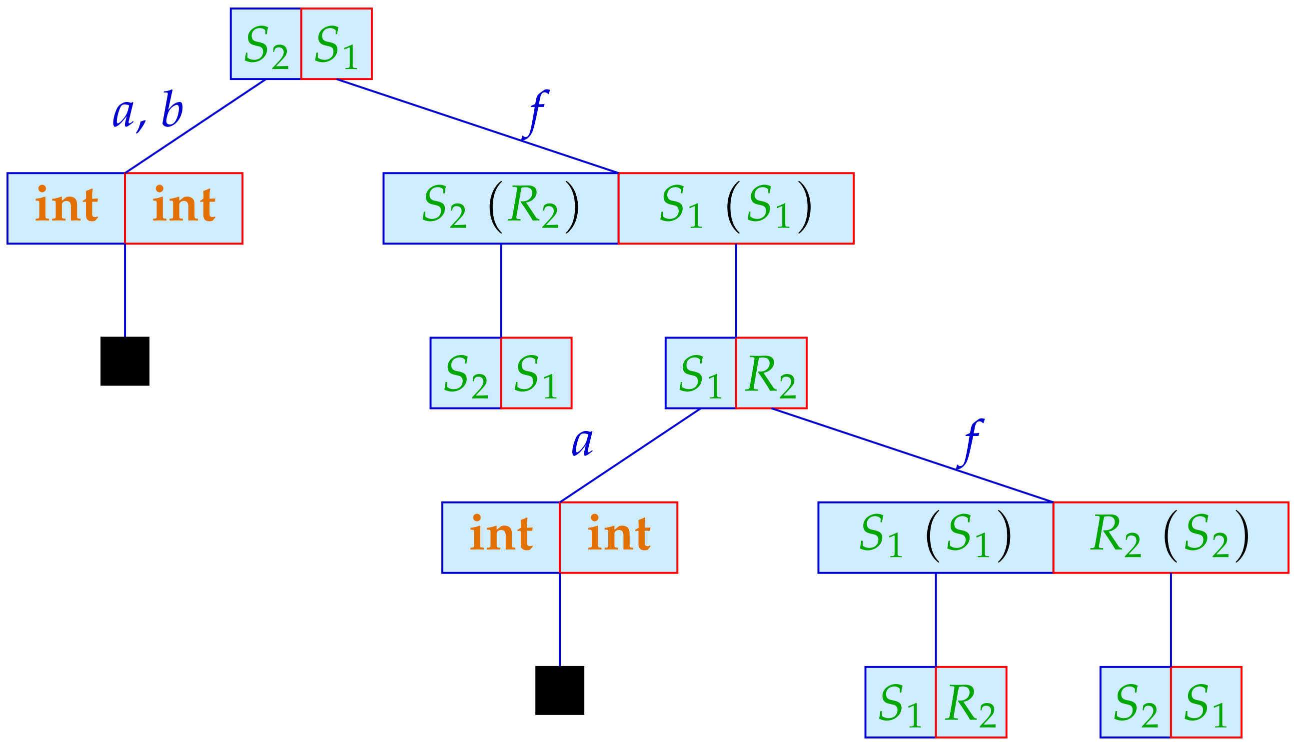
Achtung:

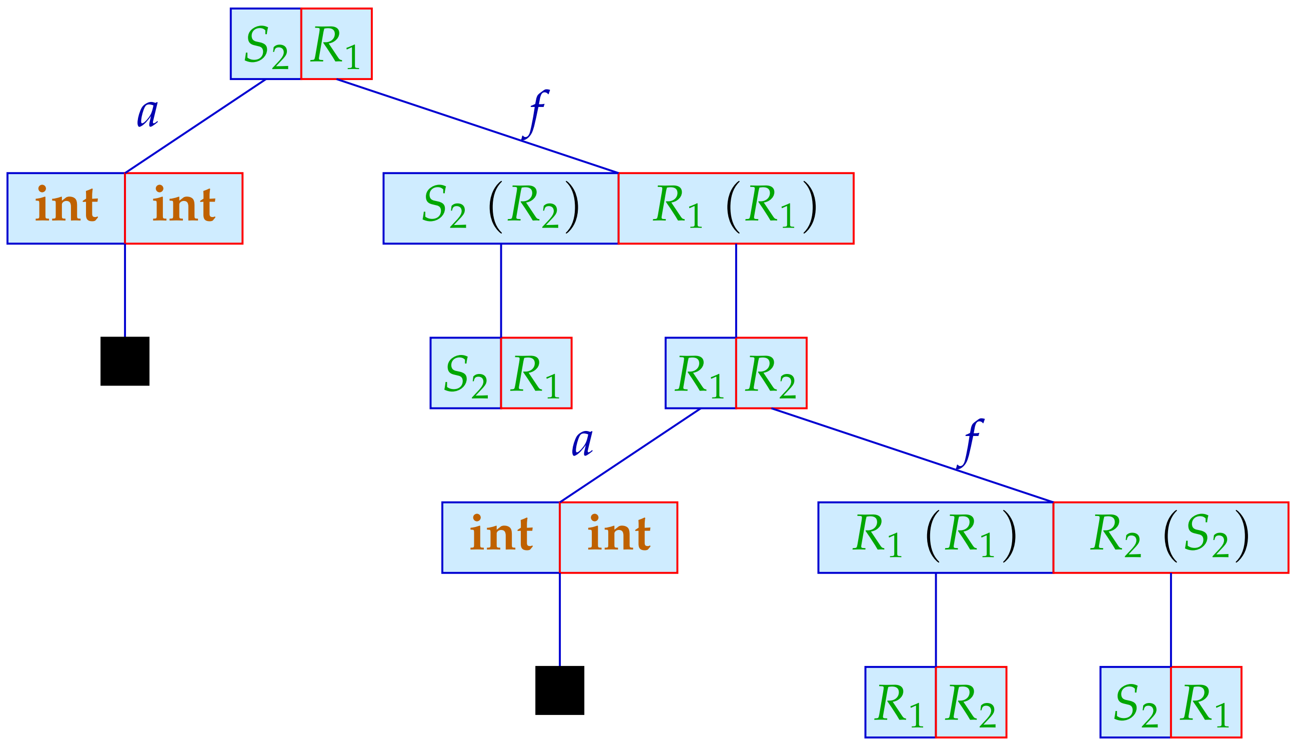
- Bei den Argumenten dreht sich die Anordnung der Typen gerade um !!!
- Diese Regeln können wir direkt benutzen, um auch für rekursive Typen die Teiltyp-Relation zu entscheiden :-)

Beispiel:

$$\begin{array}{lcl} R_1 & = & \textbf{struct}\ \{\textbf{int}\ a;\ R_1\ (R_1)\ f;\} \\ S_1 & = & \textbf{struct}\ \{\textbf{int}\ a;\ \textbf{int}\ b;\ S_1\ (S_1)\ f;\} \\ R_2 & = & \textbf{struct}\ \{\textbf{int}\ a;\ R_2\ (S_2)\ f;\} \\ S_2 & = & \textbf{struct}\ \{\textbf{int}\ a;\ \textbf{int}\ b;\ S_2\ (R_2)\ f;\} \end{array}$$




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 :-))
- Überschreiben einer Komponente mit einem spezielleren Typ ist möglich — aber nur, wenn diese keine Methode ist :-(

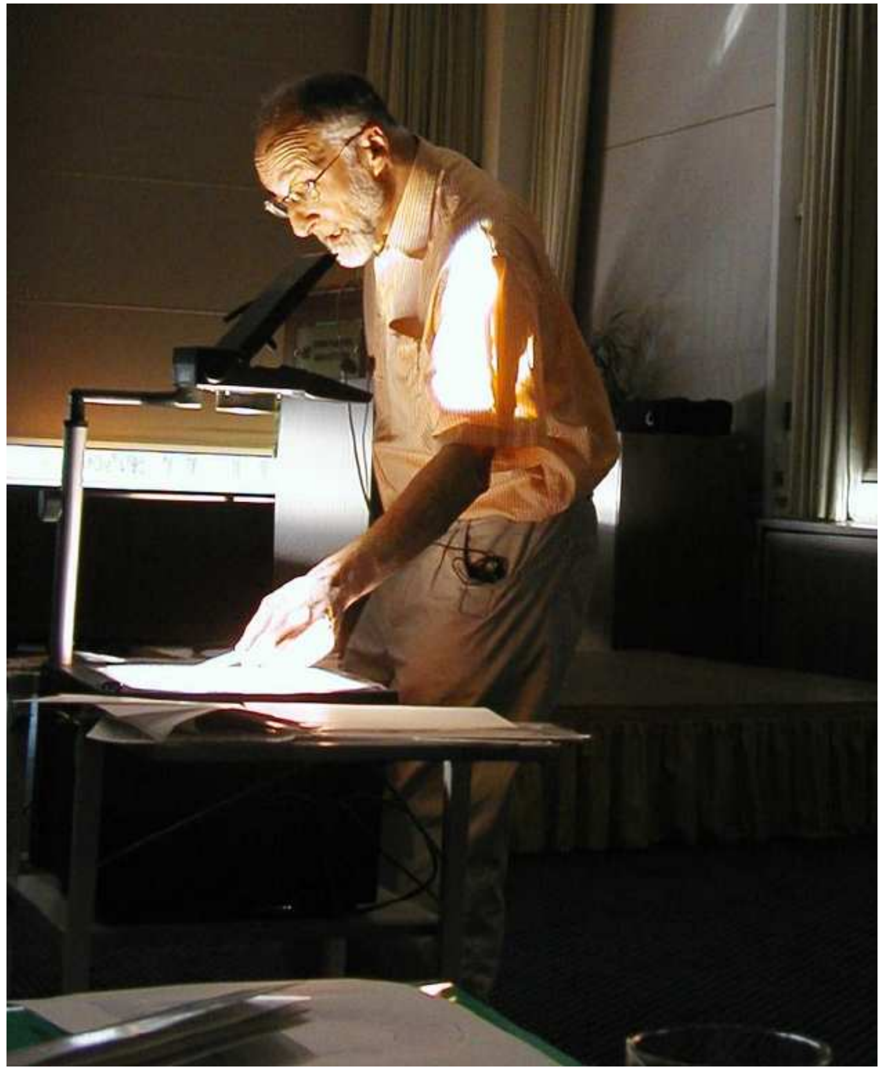
3.3 Inferieren von Typen

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

Beispiel:

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

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



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 ...

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

Beispiel:

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

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 ...

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

Wir betrachten wieder Typ-Aussagen der Form:

$$\Gamma \vdash e : t$$

Axiome:

Const: $\Gamma \vdash c : t_c$ $\qquad$ (t_c Typ der Konstante c)

Nil: $\Gamma \vdash [\,] : \mathsf{list}\ t$ $\qquad$ (t beliebig)

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

...

$\dots$

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

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

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

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

$\dots$

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

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

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

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

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

$\dots$

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

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

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

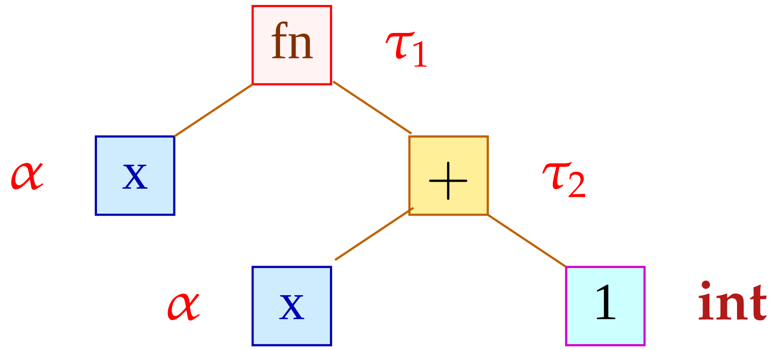
Wie raten wir die Typen der Variablen ???

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:

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



Gleichungen:

$$\begin{array}{lcl} \tau_1 & = & \alpha \longrightarrow \tau_2 \\ \tau_2 & = & \mathbf{int} \\ \alpha & = & \mathbf{int} \end{array}$$

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

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:	$e \equiv [\,]$	$\Longrightarrow$	$\tau[e] = \text{list}\,\alpha \qquad (\alpha$ 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:	$e \equiv e_1 : e_2$	$\Longrightarrow$	$\tau[e] = \tau[e_2] = \text{list}\;\tau[e_1]$
	$\ldots$		

$\ldots$

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:	$e \equiv \textbf{case } e_0 \textbf{ of } [\,] \to e_1;\ x : y \to e_2$	$\Longrightarrow$	$\tau[e_0] = \alpha[y] = \mathsf{list}\ \alpha[x]$ $\tau[e] = \tau[e_1] = \tau[e_2]$
Fun:	$e \equiv \textbf{fn } (x_1, \ldots, x_m) \Rightarrow e_1$	$\Longrightarrow$	$\tau[e] = (\alpha[x_1], \ldots, \alpha[x_m]) \to \tau[e_1]$
App:	$e \equiv e_1\ e_2$	$\Longrightarrow$	$\tau[e_1] = \tau[e_2] \to \tau[e]$
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 :-)