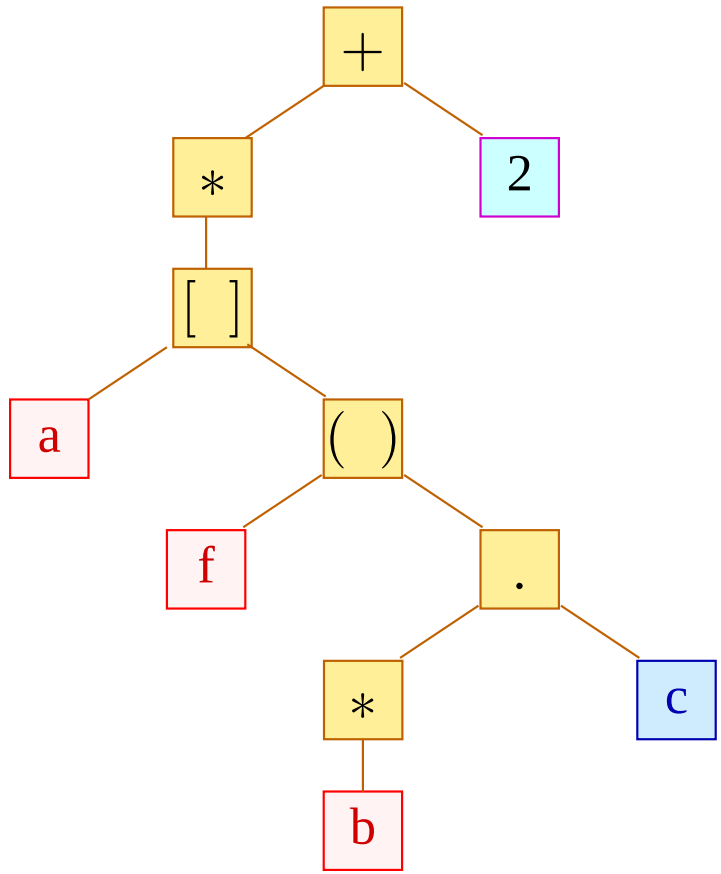
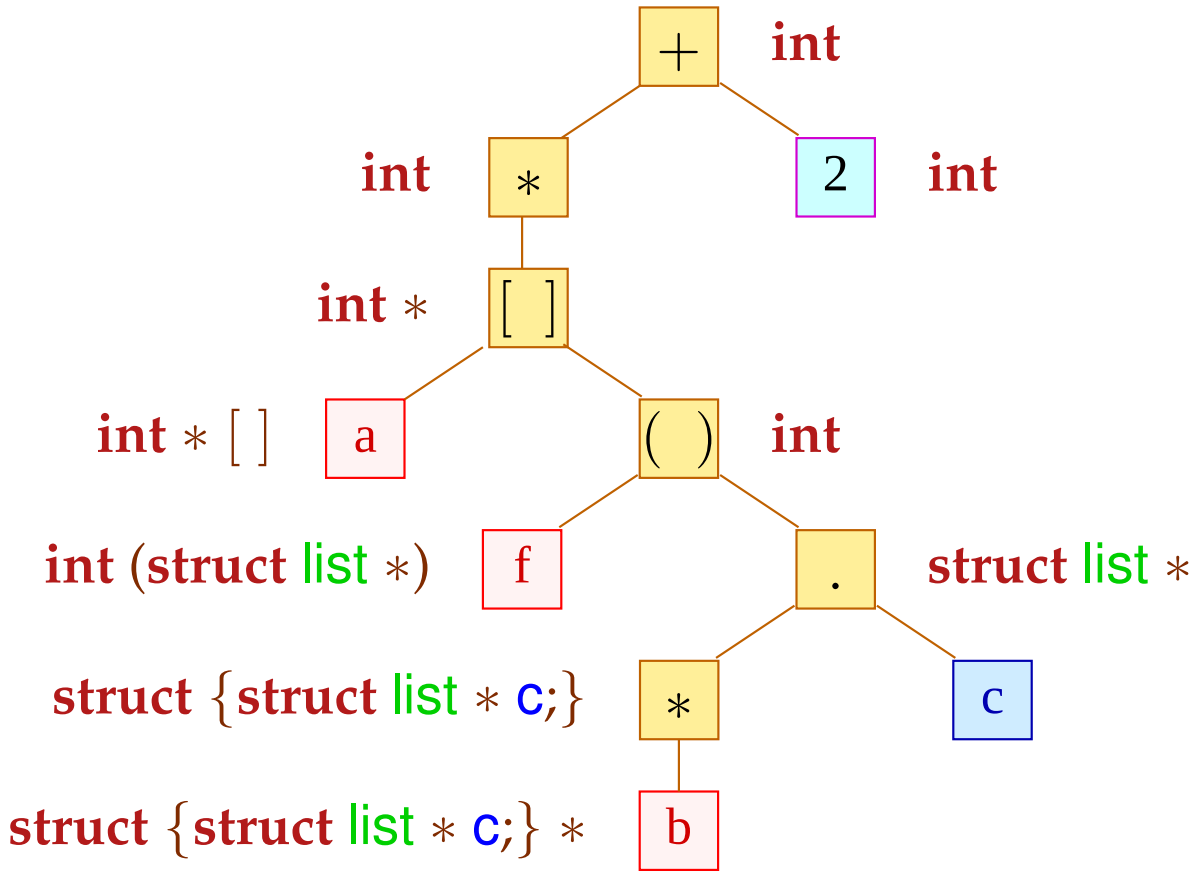
... im Beispiel:



... im Beispiel:


+
int
int
*
2
int
int *
[ ]
int * [ ]
a
( )
int
int (struct list *)
f
.
struct list *
struct {struct list * c;}
*
c
struct {struct list * c;} *
b

Diskussion:

- Welche Regel an einem Knoten angewendet werden muss, ergibt sich aus den Typen für die bereits bearbeiteten Kinderknoten :-)
- Dazu muss die Gleichheit von Typen festgestellt werden.

  Achtung:

  struct *A* {} und **struct** *B* {} werden als verschieden betrachtet !!

  Nach:

  typedef int *C*;

  bezeichnen *C* und **int** immer noch den gleichen Typ :-)
- ...

Diskussion (Forts.):

- ...
- Manche Operatoren wie z.B. $+$ sind überladen: sie besitzen mehrere verschiedene Bedeutungen.
- Welche Bedeutung ausgewählt werden soll, entscheidet sich aufgrund der Argument-Typen. Der Operator $+$ kann zum Beispiel bedeuten:

  → Addition auf **short**, **int**, **long**, **float** oder **double** :-)

  → Pointer-Arithmetik :-))

- Ist die Bedeutung ermittelt, wird (in bestimmten Fällen) für das Argument, das noch nicht den richtigen Typ hat, eine Typ-Konvertierung eingefügt.

Strukturelle Typ-Gleichheit:

Semantisch können wir zwei rekursive Typen t_1, t_2 als gleich betrachten, falls sie die gleiche Menge von Pfaden zulassen.

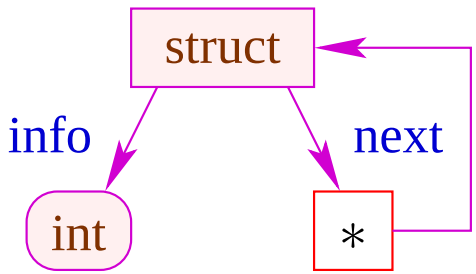
Beispiel:

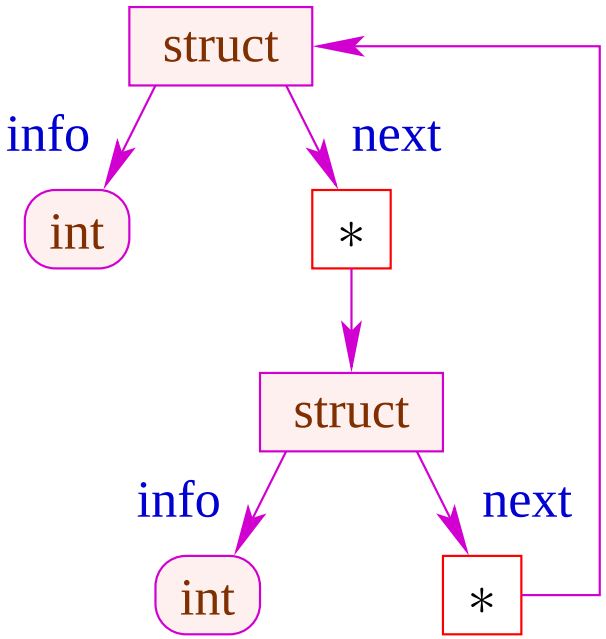
```
struct list {
    int info;
    struct list * next;
}
```

```
struct list1 {
    int info;
    struct {
        int info;
        struct list1 * next;
    } * next;
}
```

Rekursive Typen können wir als gerichtete Graphen darstellen.

... im Beispiel:





Beobachtung:

- Hat ein Knoten mehr als einen Nachfolger, tragen die ausgehenden Kanten unterschiedliche Beschriftungen :-)
- Das kann man auch für Funktions-Knoten erreichen :-)
- Der Typgraph kann damit als deterministischer endlicher Automat aufgefast werden, der alle Pfade durch den Typ akzeptiert :-))
- Zwei Typen können wir dann als äquivalent auffassen, wenn ihre Typgraphen, aufgefasst als DFAs äquivalent sind.
- Insbesondered gibt es stets einen eindeutig bestimmten minimalen Typgraphen für jeden Typ :-)
- Strukturelle Äquivalenz rekursiver Typen ist deshalb schnell entscheidbar !!!

Alternativer Algorithmus:

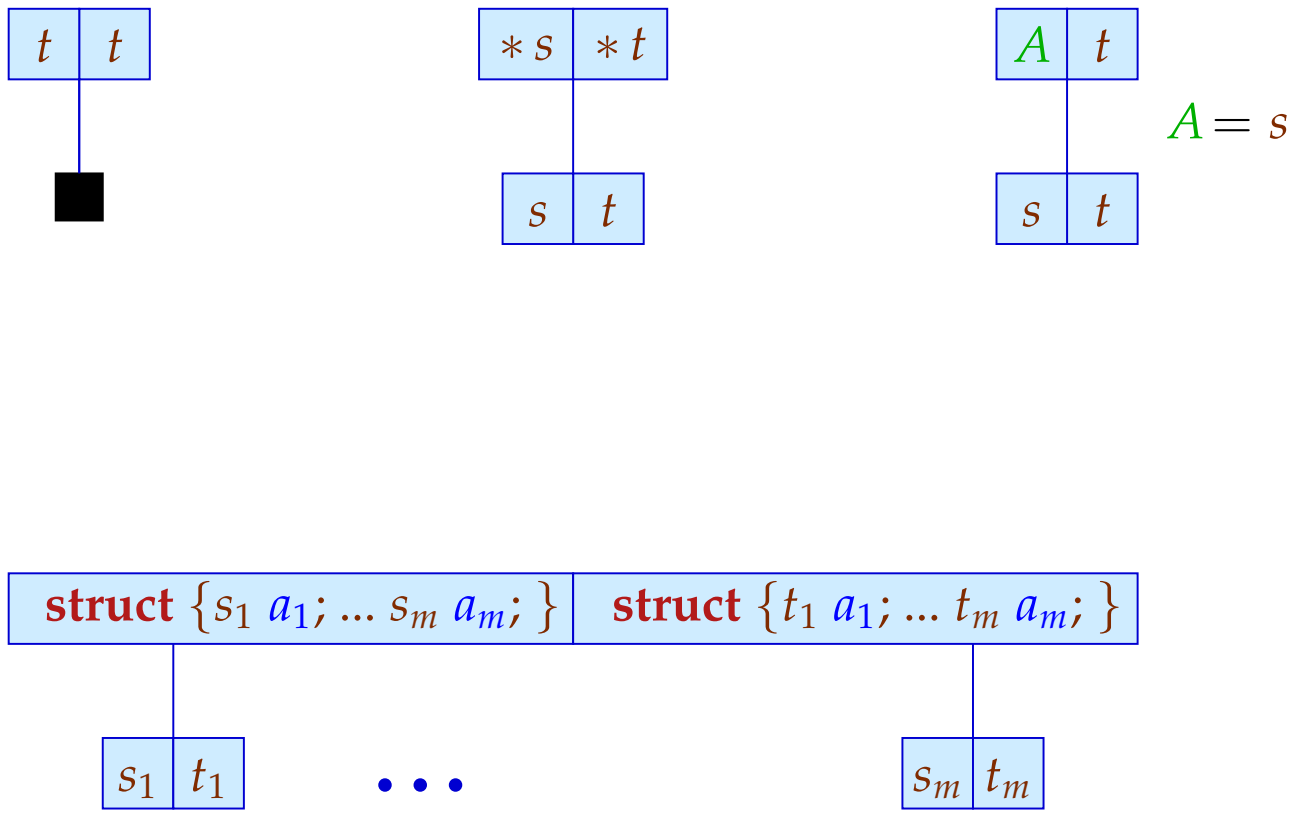
Idee:

- Verwalte Äquivalenz-Anfragen für je zwei Typausdrücke ...
- Sind die beiden Ausdrücke syntaktisch gleich, ist alles gut :-)
- Andernfalls reduziere die Äquivalenz-Anfrage zwischen Äquivalenz-Anfragen zwischen (hoffentlich einfacheren anderen Typausdrücken :-)

Nehmen wir an, rekursive Typen würden mithilfe von Typ-Gleichungen der Form:

$$A = t$$

eingeführt ...


t t
∗ s ∗ t
s t
A t
A = s
s t
struct {s1 a1; ... sm am; }
struct {t1 a1; ... tm am; }
s1 t1
• • •
sm tm

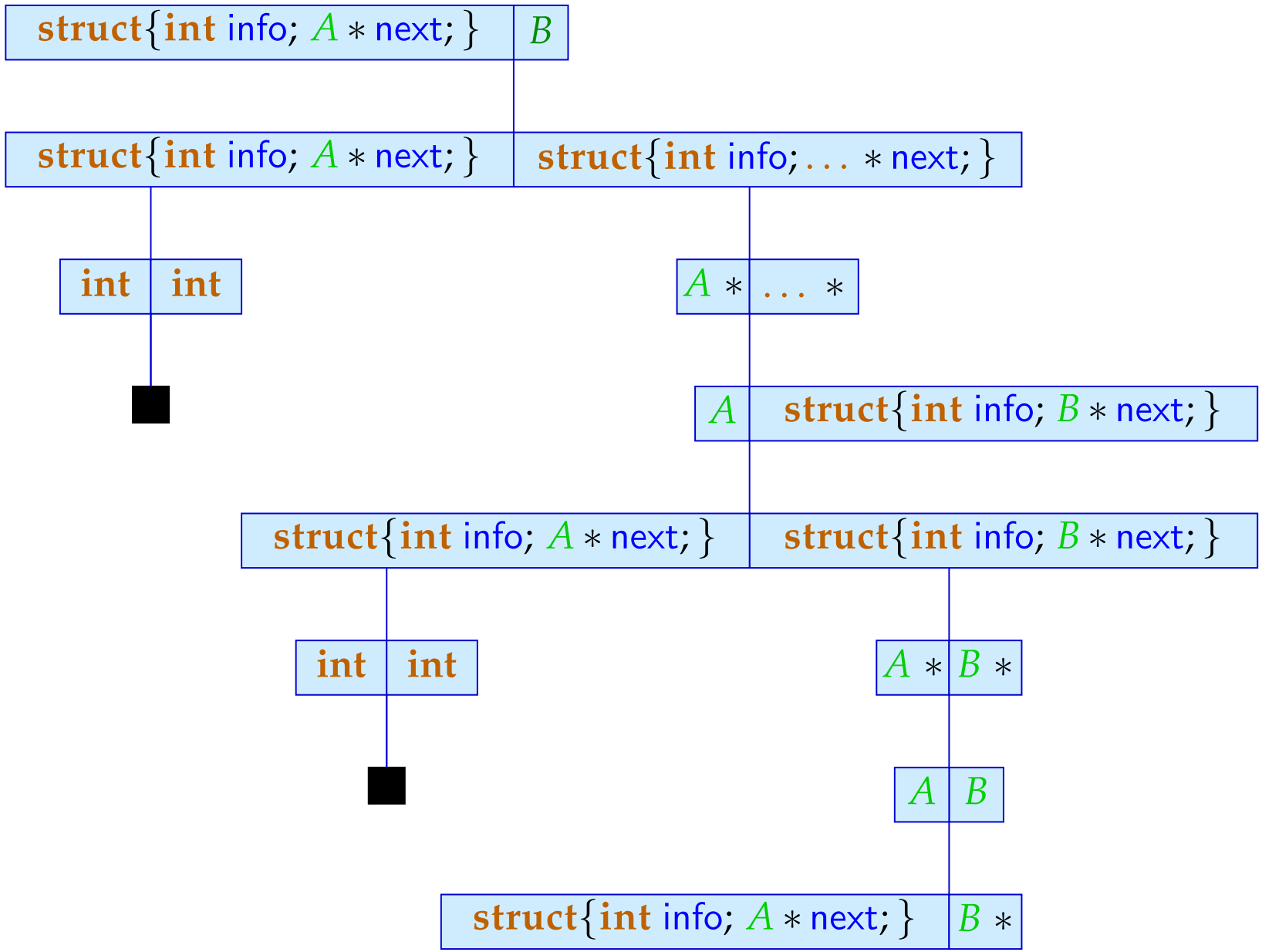
... im Beispiel:

$$
\begin{aligned}
A &= \textbf{struct}\ \{\textbf{int}\ \mathsf{info};\ A * \mathsf{next};\} \\
B &= \textbf{struct}\ \{\textbf{int}\ \mathsf{info}; \\
&\quad\ \textbf{struct}\ \{\textbf{int}\ \mathsf{info};\ B * \mathsf{next};\} * \mathsf{next};\}
\end{aligned}
$$

Wir fragen uns etwa, ob gilt:

$$\textbf{struct}\ \{\textbf{int}\ \mathsf{info};\ A * \mathsf{next};\} = B$$

Dazu konstruieren wir:


struct{int info; A * next; }
B
struct{int info; A * next; }
struct{int info; . . . * next; }
int
int
A *
. . . *
A
struct{int info; B * next; }
struct{int info; A * next; }
struct{int info; B * next; }
int
int
A *
B *
A
B
struct{int info; A * next; }
B *

Diskussion:

- Stoßen wir bei der Konstruktion des Beweisbaums auf eine Äquivalenz-Anfrage, auf die keine Regel anwendbar ist, gibt es einen Widerspruch !!!
- Die Konstruktion des Beweisbaums kann dazu führen, dass die gleiche Äquivalenz-Anfrage ein weiteres Mal auftritt ...
- Taucht eine Äquivalenz-Anfrage ein weiteres Mal auf, können wir hier abbrechen ;-)

$\Longrightarrow$ die Anzahl zu betrachtender Anfragen ist endlich :-)

$\Longrightarrow$ das Verfahren terminiert :-))

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 ...

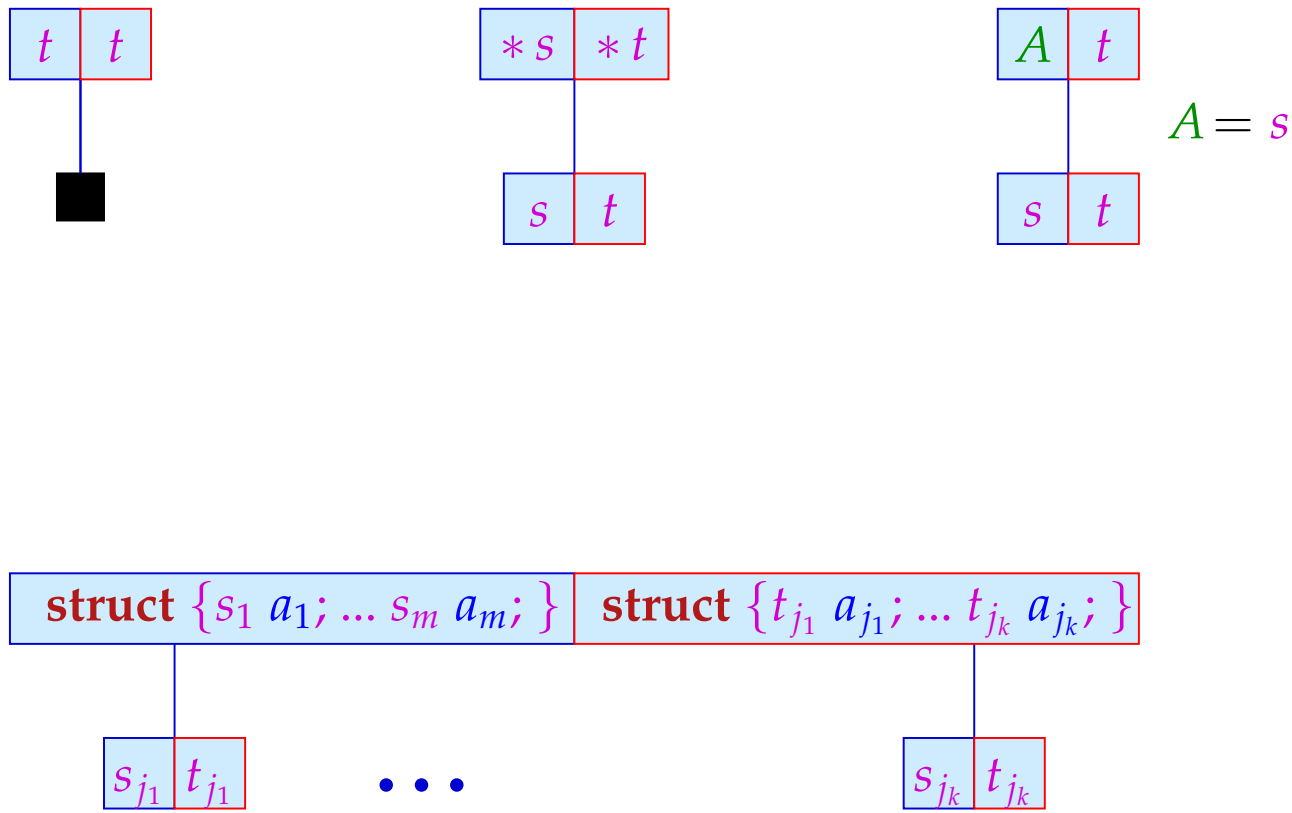
$\Longrightarrow$

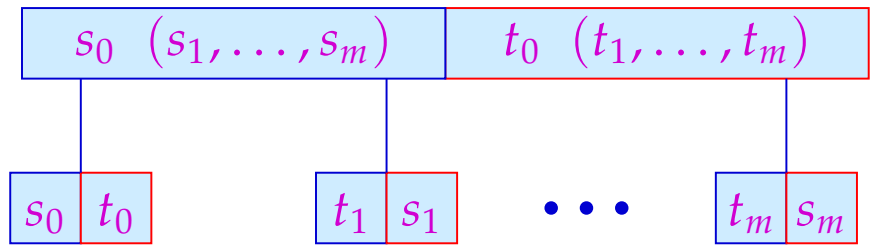
Erweitere Teiltypen-Beziehungen der Basistypen auf komplexe Typen :-)

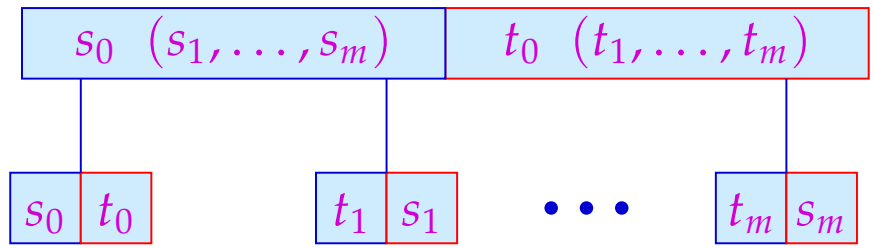
Beispiel:

```
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
$A = s$
s t
struct $\{s_1\ a_1;\ \ldots\ s_m\ a_m;\}$
struct $\{t_{j_1}\ a_{j_1};\ \ldots\ t_{j_k}\ a_{j_k};\}$
s_{j_1} t_{j_1}
$\cdots$
s_{j_k} t_{j_k}

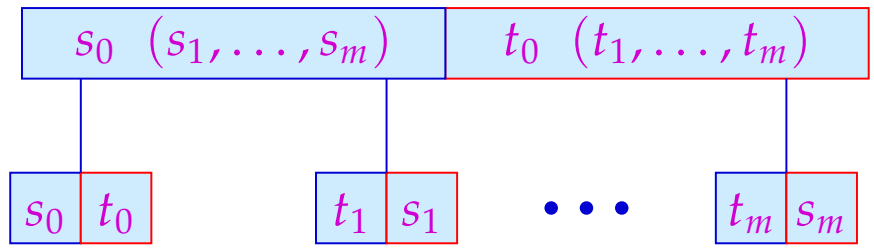

$s_0 \ (s_1, \ldots, s_m)$
$t_0 \ (t_1, \ldots, t_m)$
s_0
t_0
t_1
s_1
t_m
s_m



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:

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

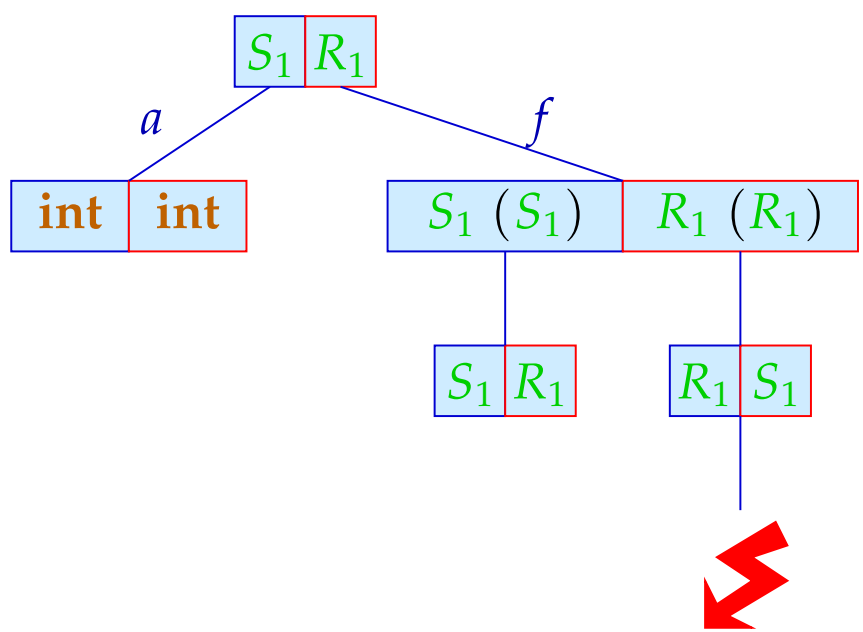
$$\textbf{int}\ (\textbf{int}) \quad \not\leq \quad \textbf{float}\ (\textbf{float})$$

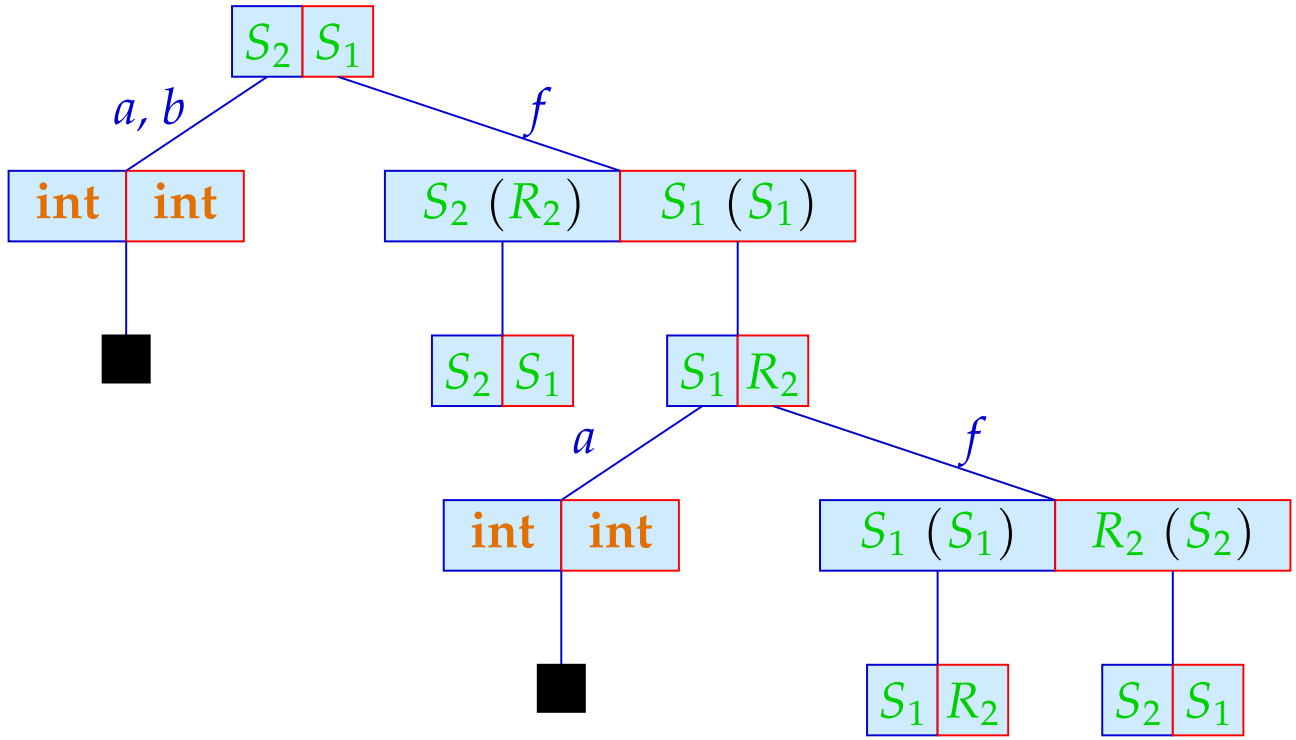
Achtung:

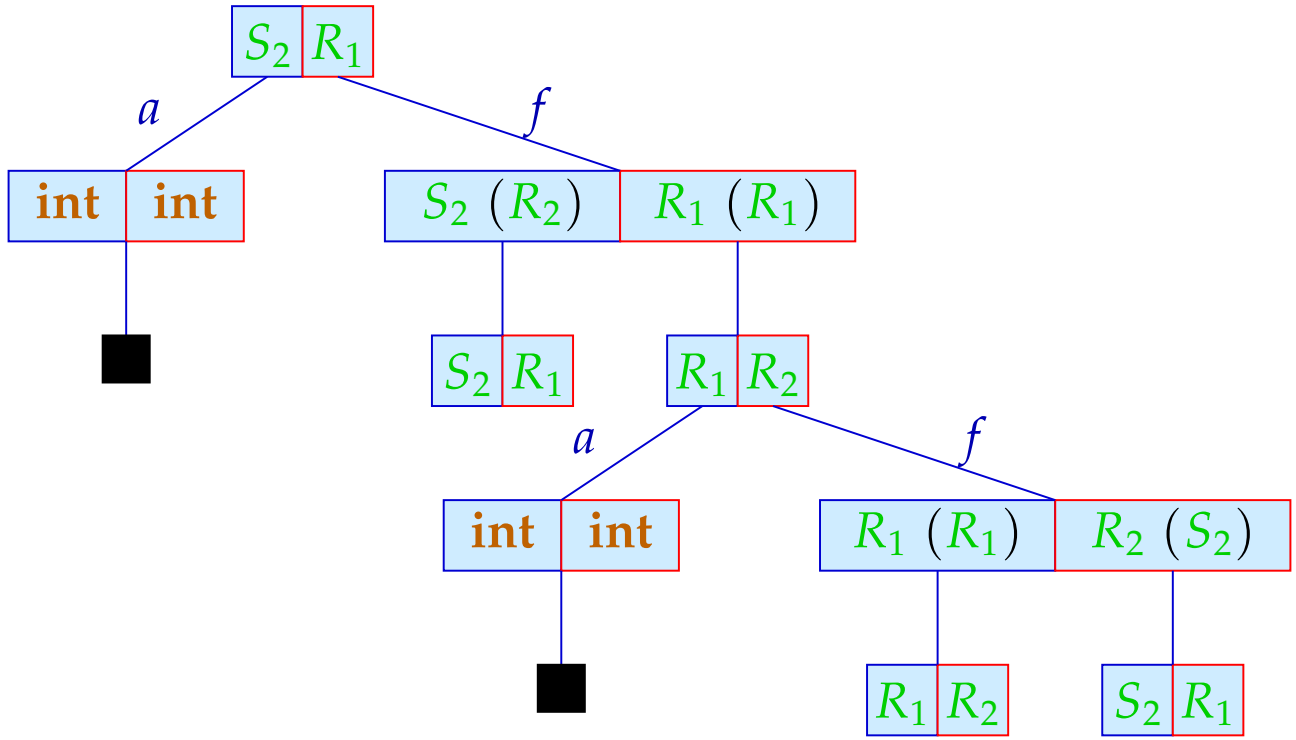
- Bei den Argumenten dreht sich die Anordnung der Typen gerade um !!!
- Diese Regeln können wir direkt benutzen, um auch für rekursive Typen die Teiltyp-Relation zu entscheiden :-)

Beispiel:

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




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 :-(

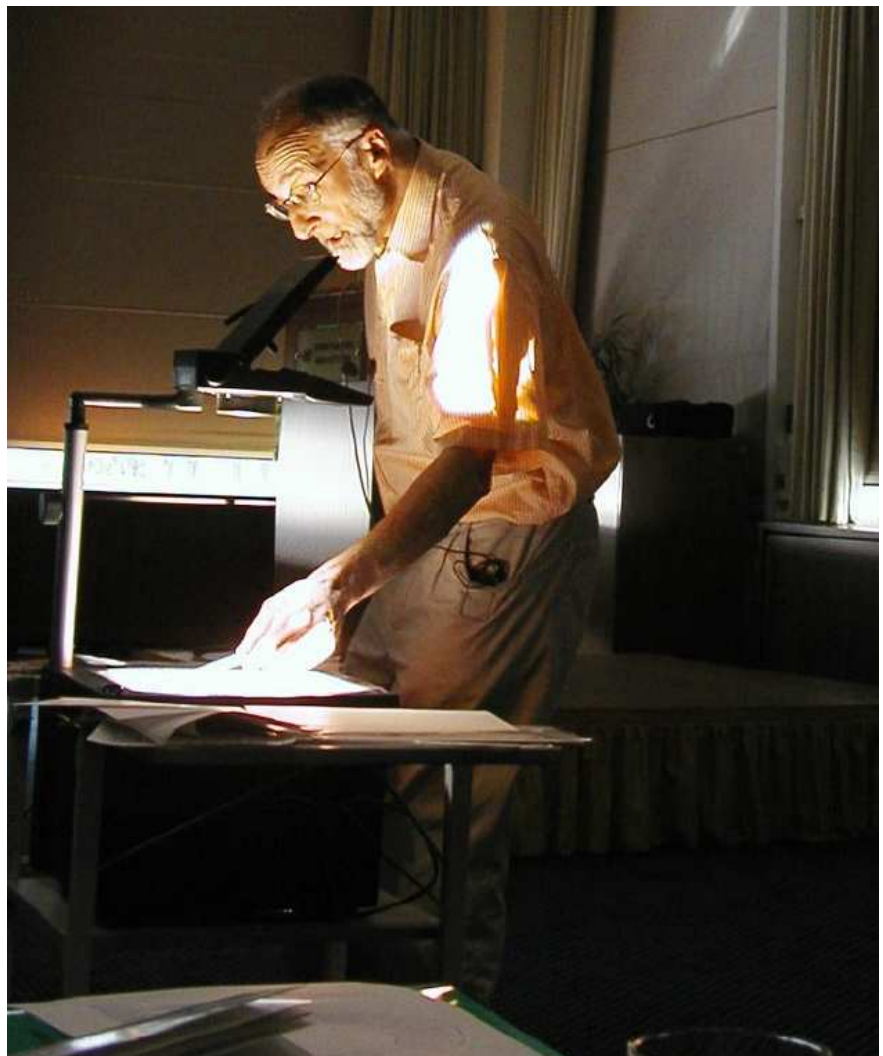
3.3 Inferieren von Typen

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

Beispiel:

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

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



Robin (Dumbledore) Milner, Edinburgh