Array: $$\frac{\Gamma \vdash e_1 : t* \qquad \Gamma \vdash e_2 : \textbf{int}}{\Gamma \vdash e_1[e_2] : t}$$

Array: $$\frac{\Gamma \vdash e_1 : t\,[\,] \qquad \Gamma \vdash e_2 : \textbf{int}}{\Gamma \vdash e_1[e_2] : t}$$

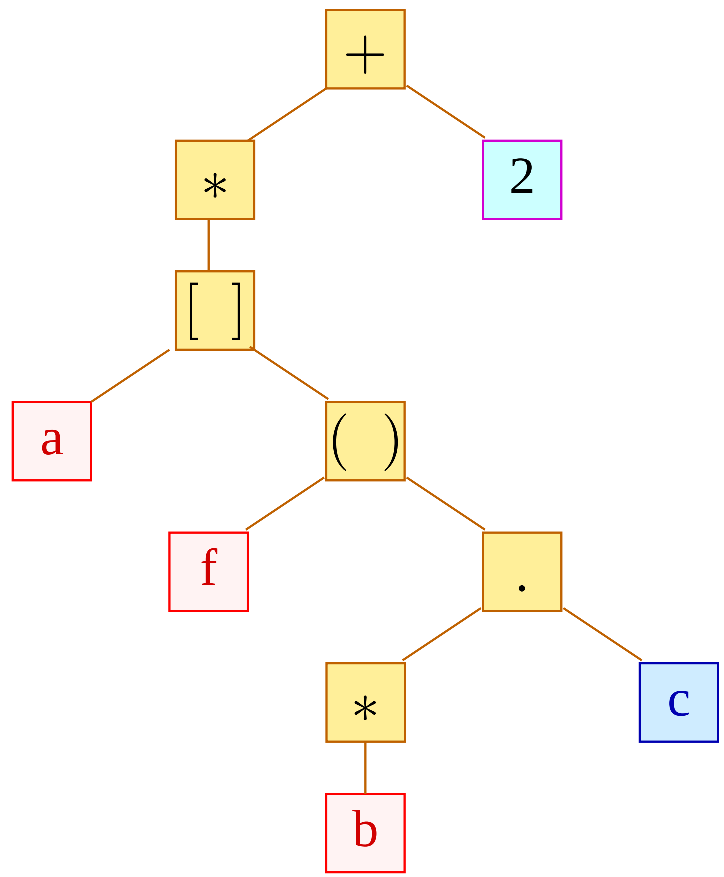
Struct: $$\frac{\Gamma \vdash e : \textbf{struct}\ \{t_1\ a_1; \ldots t_m\ a_m;\}}{\Gamma \vdash e.a_i : t_i}$$

App: $$\frac{\Gamma \vdash e : t\,(t_1, \ldots, t_m) \qquad \Gamma \vdash e_1 : t_1 \quad \ldots \quad \Gamma \vdash e_m : t_m}{\Gamma \vdash e(e_1, \ldots, e_m) : t}$$

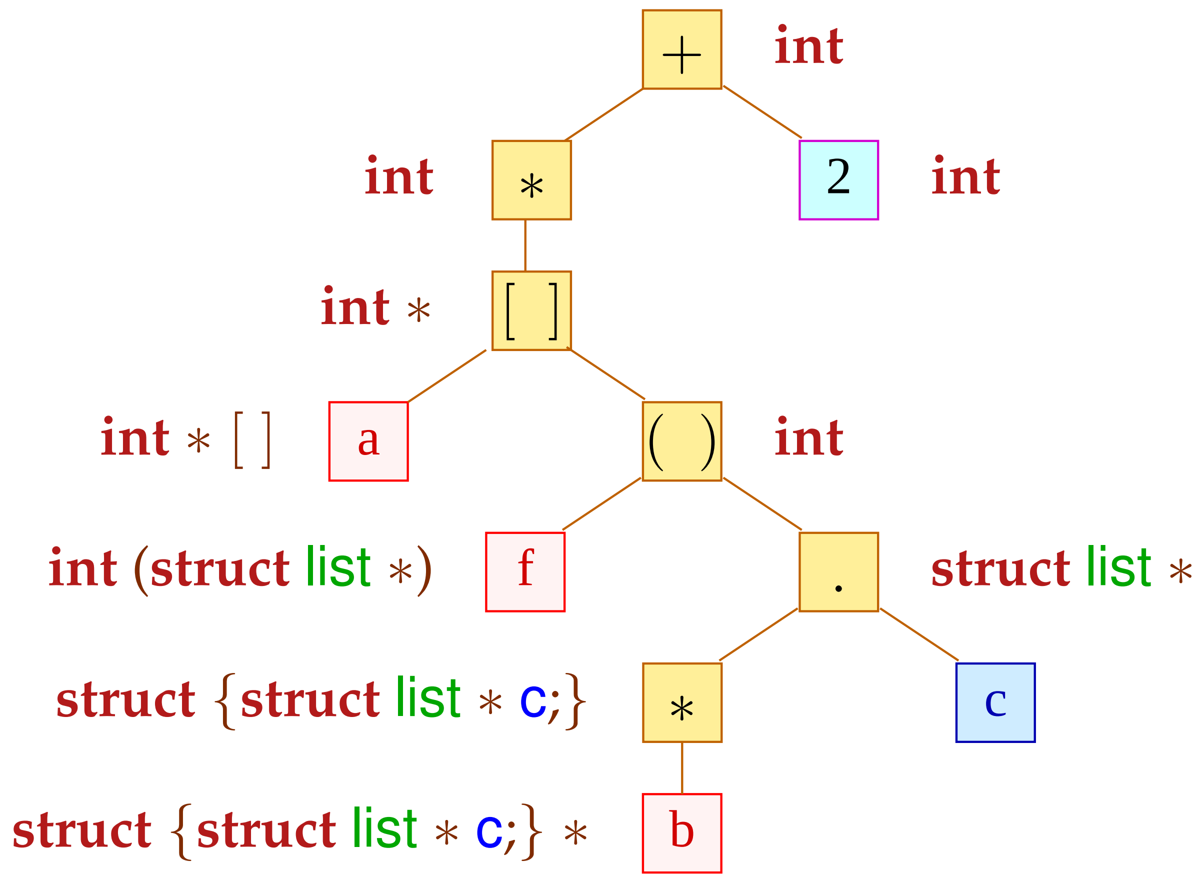
Op: $$\frac{\Gamma \vdash e_1 : \textbf{int} \qquad \Gamma \vdash e_2 : \textbf{int}}{\Gamma \vdash e_1 + e_2 : \textbf{int}}$$

Cast: $$\frac{\Gamma \vdash e : t_1 \qquad t_1 \text{ in } t_2 \text{ konvertierbar}}{\Gamma \vdash (t_2)\ e : t_2}$$

... im Beispiel:



... im Beispiel:



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

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:

  $\longrightarrow$ Addition auf **short**, **int**, **long**, **float** oder **double** :-)

  $\longrightarrow$ 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.

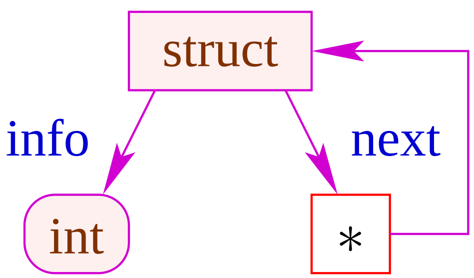
Beispiel:

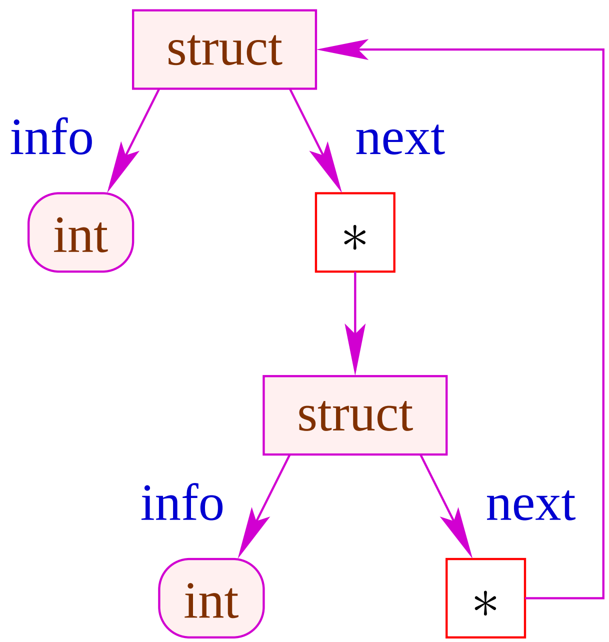
```
struct list {
    int info;
    struct list * next;
}
```

```
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 aufgefasst werden, der alle Pfade durch den Typ akzeptiert :-))
- Zwei Typen können wir dann als äquivalent auffassen, wenn ihre Typgraphen, aufgefasst als DFAs äquivalent sind.
- Insbesondere gibt es stets einen eindeutig bestimmten minimalen Typgraphen für jeden Typ :-)
- Strukturelle Äquivalenz rekursiver Typen ist deshalb schnell entscheidbar !!!

Alternativer Algorithmus:

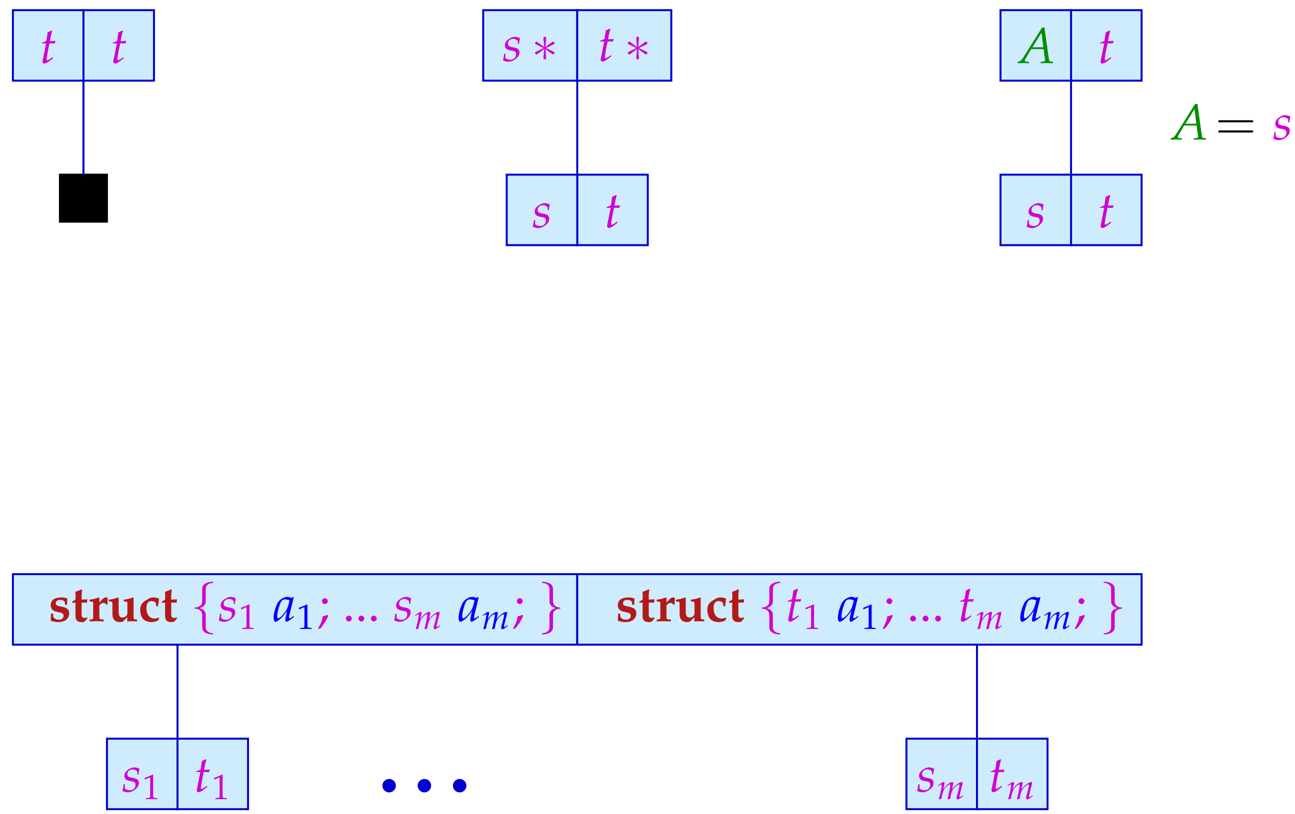
Idee:

- 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

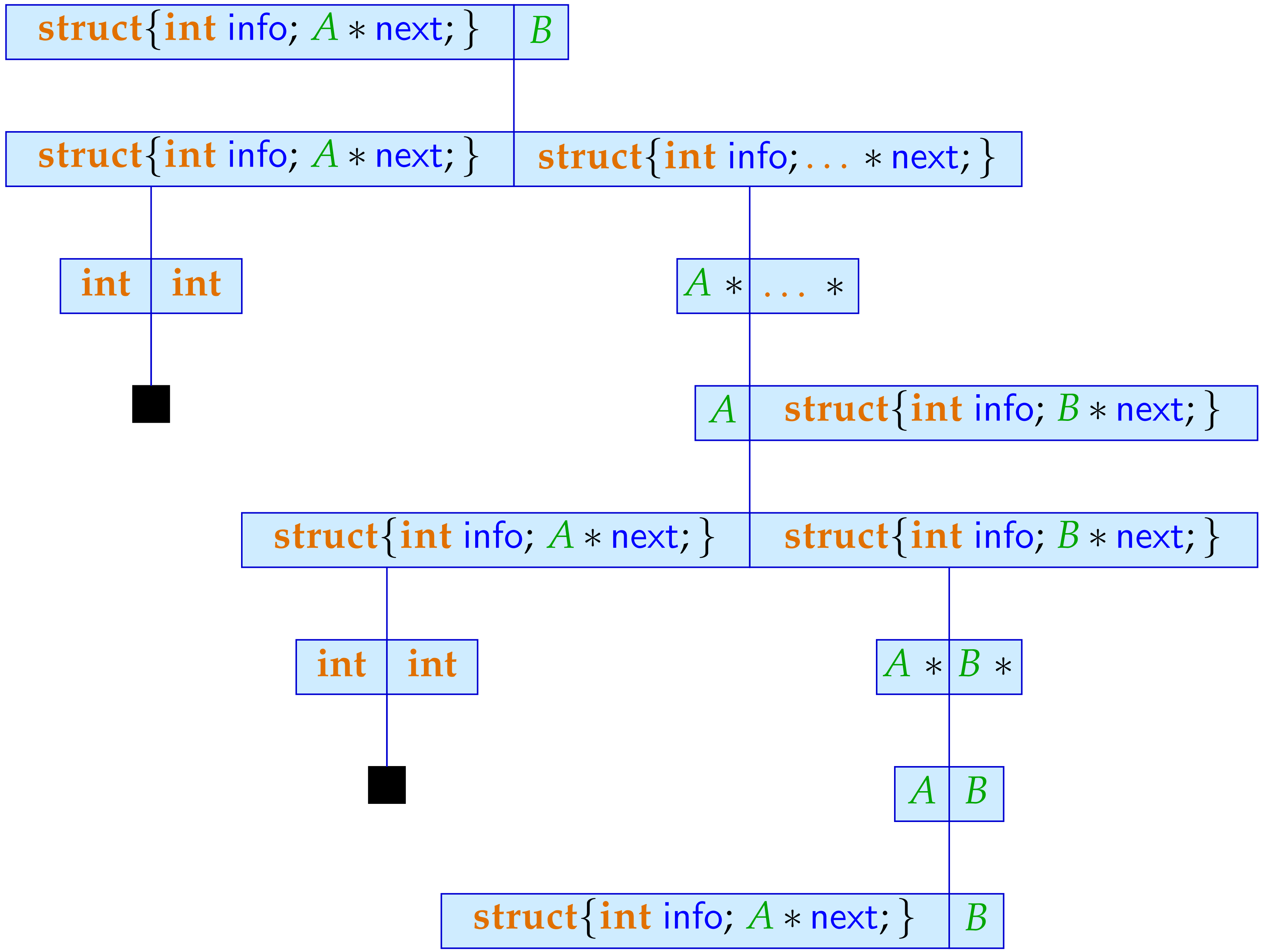
... im Beispiel:

$$\begin{array}{lll} A & = & \mathbf{struct}\ \{\mathbf{int}\ \mathsf{info};\ A * \mathsf{next};\} \\ B & = & \mathbf{struct}\ \{\mathbf{int}\ \mathsf{info}; \\ & & \mathbf{struct}\ \{\mathbf{int}\ \mathsf{info};\ B * \mathsf{next};\} * \mathsf{next};\} \end{array}$$

Wir fragen uns etwa, ob gilt:

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

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 ...

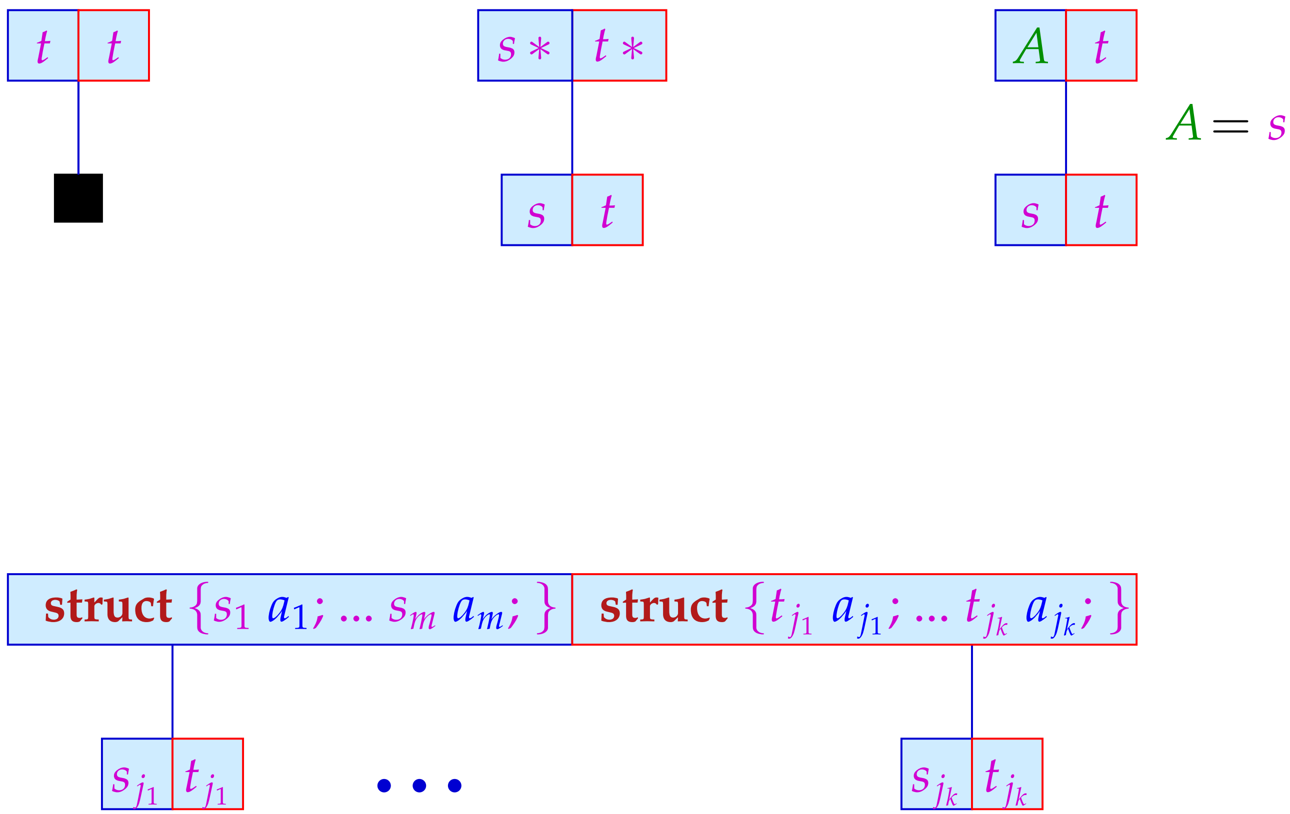
$\Longrightarrow$

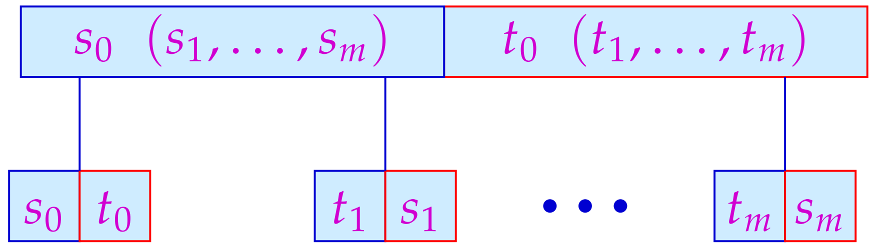
Erweitere Teiltypen-Beziehungen der Basistypen auf komplexe Typen :-)

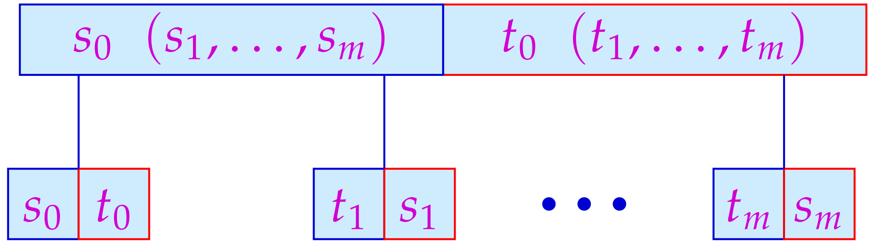
Beispiel:

```
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 ...

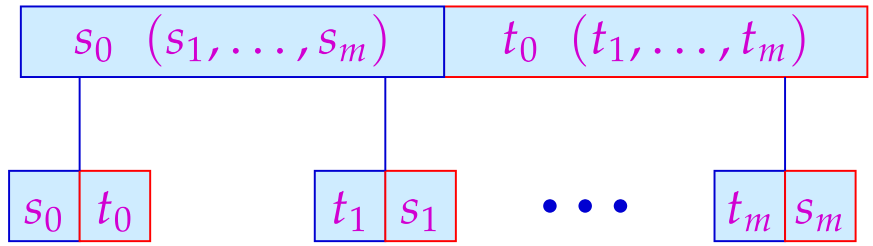

t t
s * t *
s t
A t
A = s
s t
struct {s1 a1; ... sm am; } struct {tj1 aj1; ... tjk ajk; }
sj1 tj1
sjk tjk


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



Beispiele:

$$
\begin{array}{lcl}
\textbf{struct}\ \{\textbf{int}\ a;\ \ \textbf{int}\ b;\} & \leq & \textbf{struct}\ \{\textbf{float}\ a;\} \\
\textbf{int}\ (\textbf{int}) & \not\leq & \textbf{float}\ (\textbf{float})
\end{array}
$$



Beispiele:

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

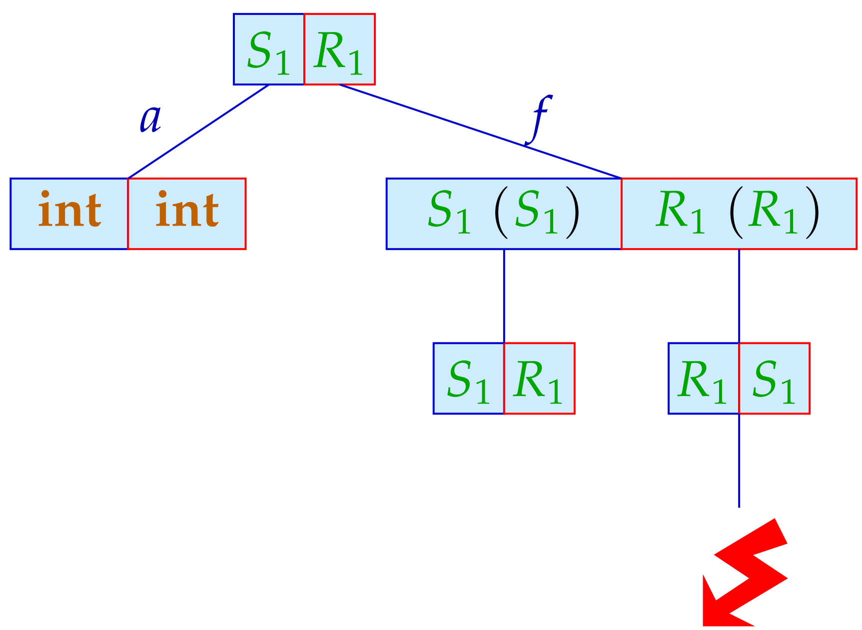
$$\textbf{int}\ (\textbf{int}) \quad \not\leq \quad \textbf{float}\ (\textbf{float})$$

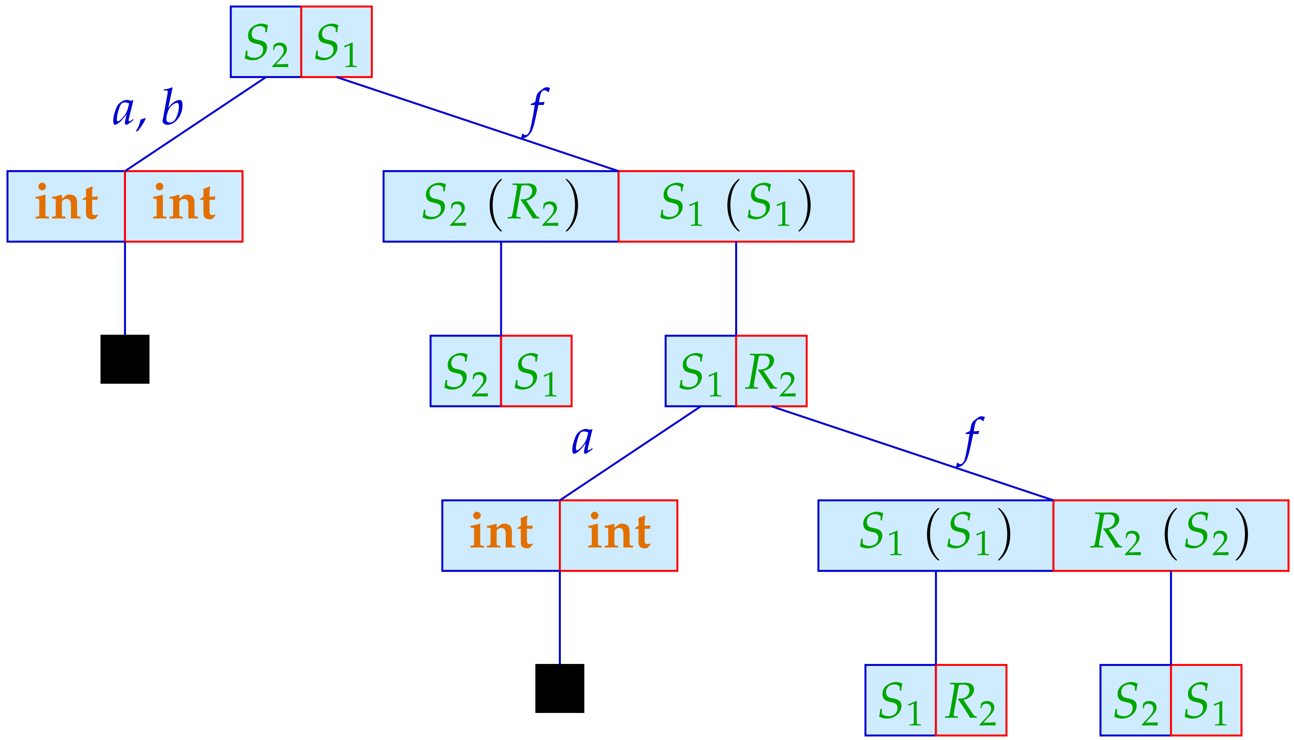
Achtung:

- 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 :-)

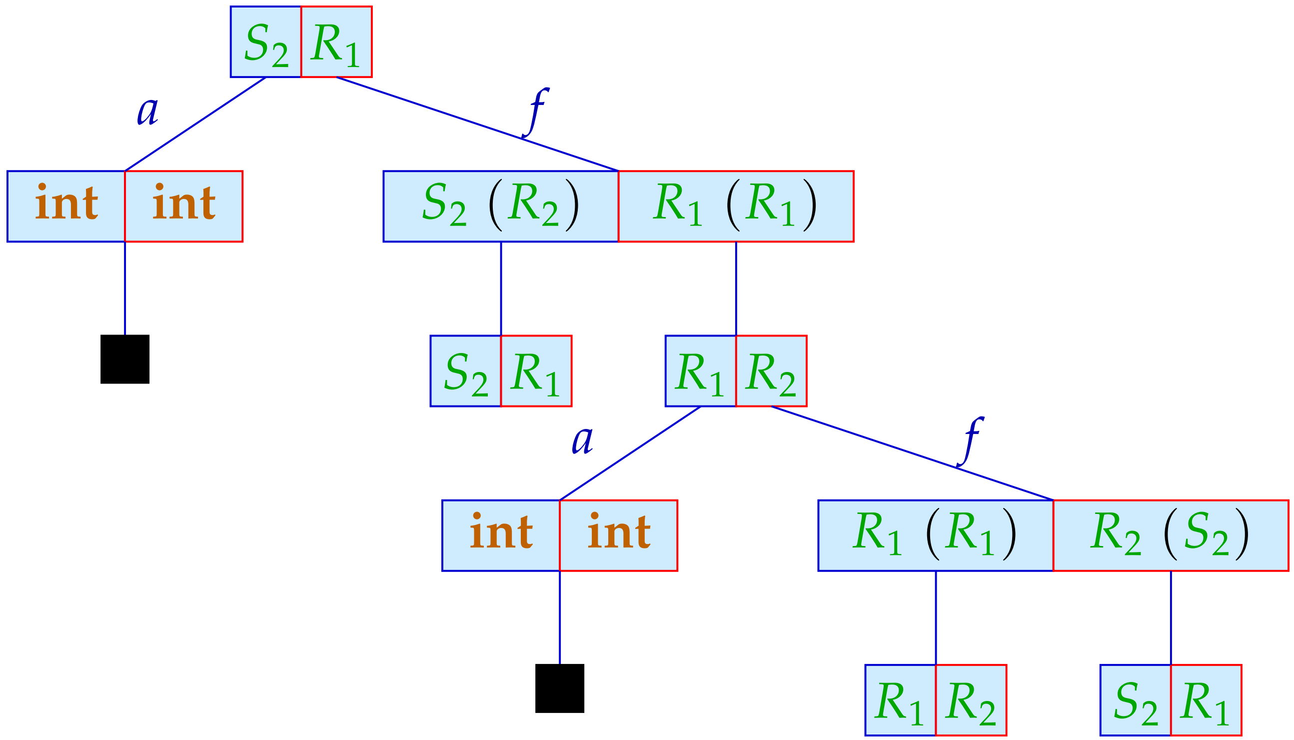
Beispiel:

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





S2 S1
a, b
f
int int
S2 (R2) S1 (S1)
S2 S1
S1 R2
a
f
int int
S1 (S1) R2 (S2)
S1 R2
S2 S1


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:

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

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