Typen werden durch Typ-Ausdrücke beschrieben.

Die Menge T der Typausdrücke enthält:

(1) Basis-Typen: **int**, **boolean**, **float**, **void** ...

(2) Typkonstruktoren, angewendet auf Typen, z.B.:

- Verbunde: **struct** $\{ t_1\ a_1; \ldots t_k\ a_k; \}$
- Zeiger: $t\ *$
- Felder: $t\ []$

  Achtung:
  In C muss zusätzlich eine Größe spezifiziert werden; die Variable muss dann zwischen t und $[n]$ stehen :-(
- Funktionen: $t\ (t_1, \ldots, t_k)$

  Achtung:
  In C muss die Variable zwischen t und $(t_1, \ldots, t_k)$ stehen.

  In SML dagegen würde man diesen Typ anders herum schreiben:
  $t_1 * \ldots * t_k \rightarrow t$:-)

  Wir benutzen: $(t_1, \ldots, t_k)$ als Tupel-Typen.

(3) Typ-Namen.

Typ-Namen sind nützlich:

- als Abkürzung :-)

  In C kann man diese mithilfe von **typedef** einführen:

  typedef t x ;

- zur Konstruktion rekursiver Typen ...

Beispiel:

```
struct list0 {
        int info;
        struct list1 * next;
    } ;
```

```
struct list1 {
        int info;
        struct list0 * next;
    } ;
```

Aufgabe:

Gegeben: eine Menge von Typ-Deklarationen $\Gamma = \{t_1\ x_1; \ldots t_m\ x_m;\}$

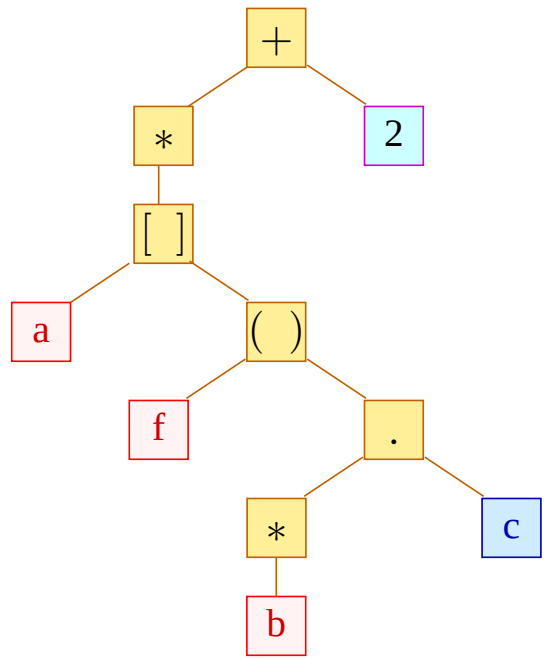
Überprüfe: Kann ein Ausdruck e mit dem Typ t versehen werden?

Beispiel:

```
struct list {int info; struct list *next;};
int f(struct list *l) {return 1;};
struct {struct list *c;} *b;
int *a[11];
```

Betrachte den Ausdruck:

$$*\mathsf{a}[f(\mathsf{b} \rightarrow \mathsf{c})] + 2;$$



Idee:

- Traversiere den Syntaxbaum bottom-up.
- Für Bezeichner sagt uns Γ den richtigen Typ :-)
- Konstanten wie 2 oder 0.5 sehen wir den Typ direkt an ;-)
- Die Typen für die inneren Knoten erschießen wir mithilfe von Typ-Regeln.

Formal betrachten wir Aussagen der Form:

$$\Gamma \vdash e : t$$

// (In der Typ-Umgebung Γ hat e den Typ t)

Axiome:

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

Var: $\Gamma \vdash x : \Gamma(x)$ (x Variable)

Regeln:

Ref: $$\frac{\Gamma \vdash e : t}{\Gamma \vdash \& e : t *}$$

Deref: $$\frac{\Gamma \vdash e : t *}{\Gamma \vdash * e : t}$$

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:

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

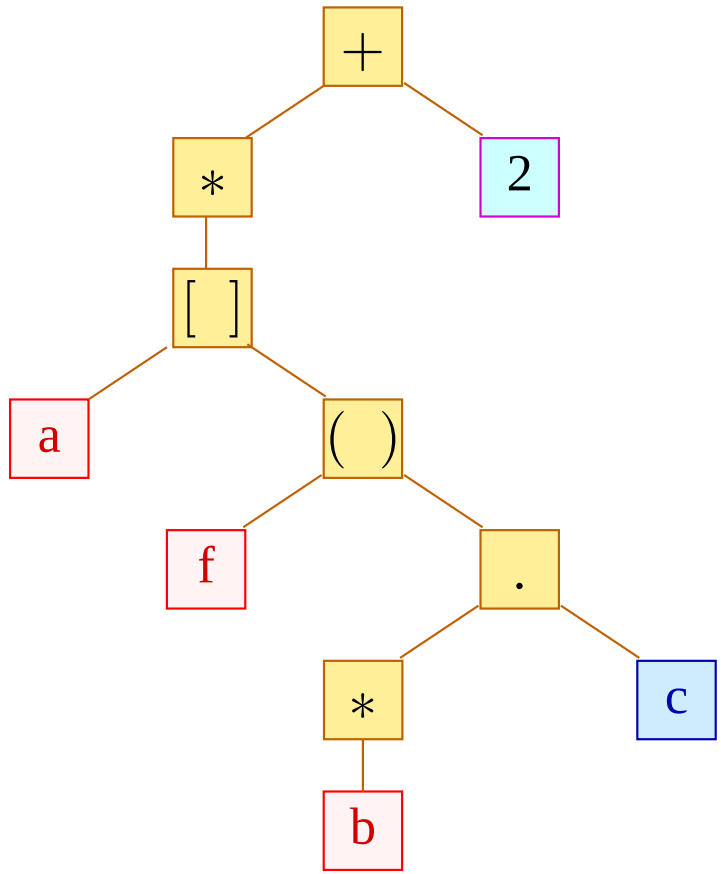
Op:

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

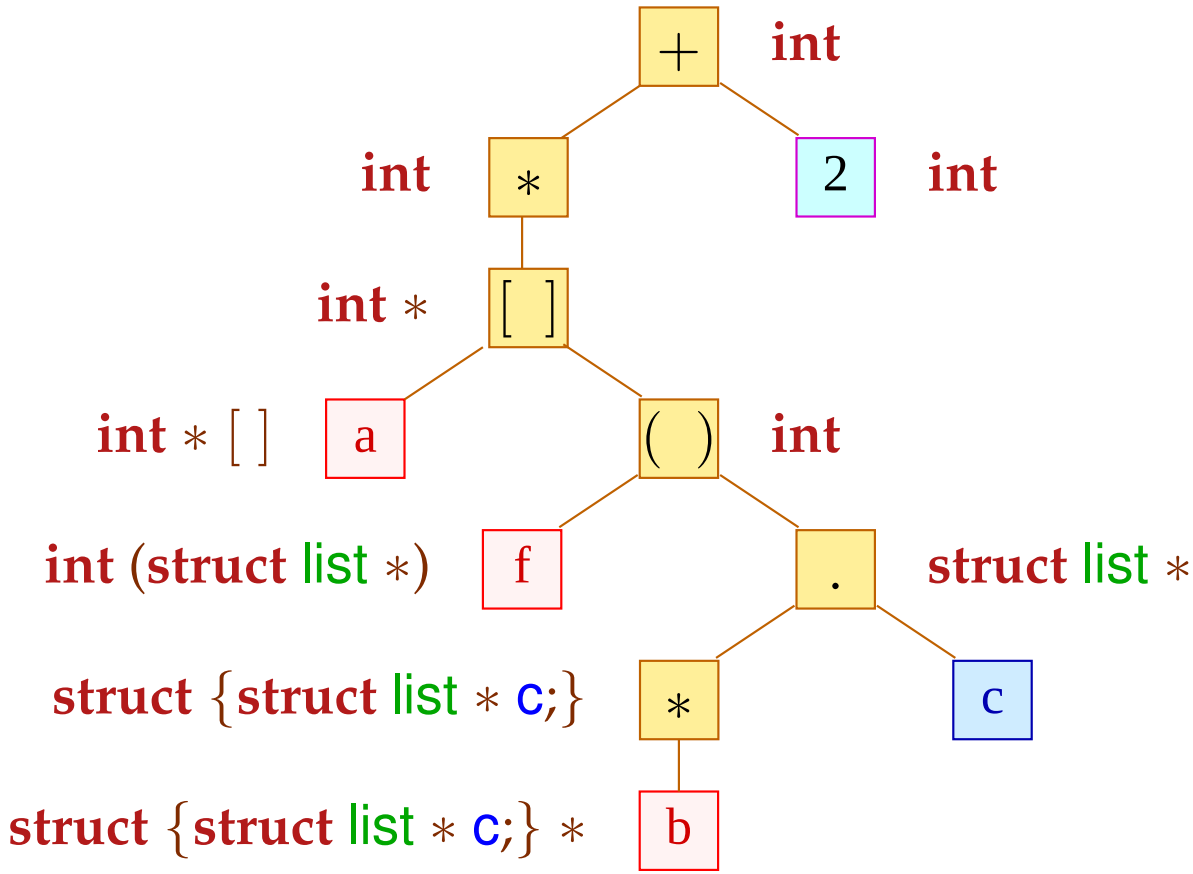
Cast:

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

... im Beispiel:



... im Beispiel:


+
int
int
*
2
int
int *
[ ]
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.

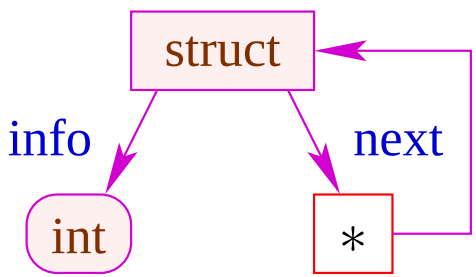
Beispiel:

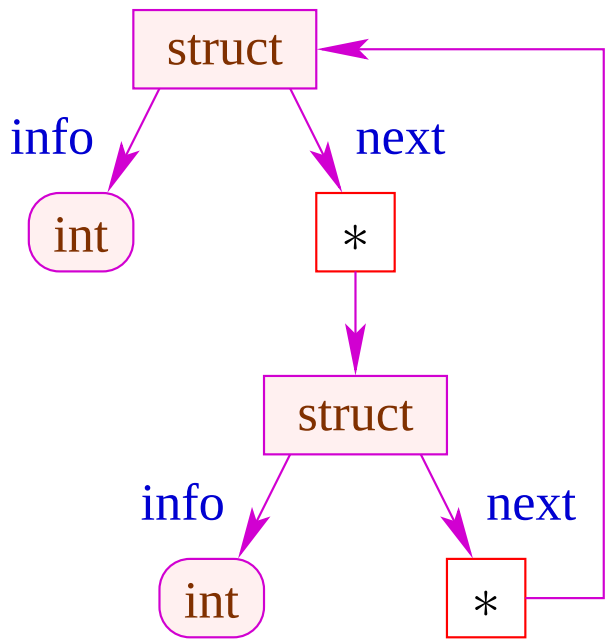
```
struct list {
    int info;
    struct list * next;
}
```

```
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 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:

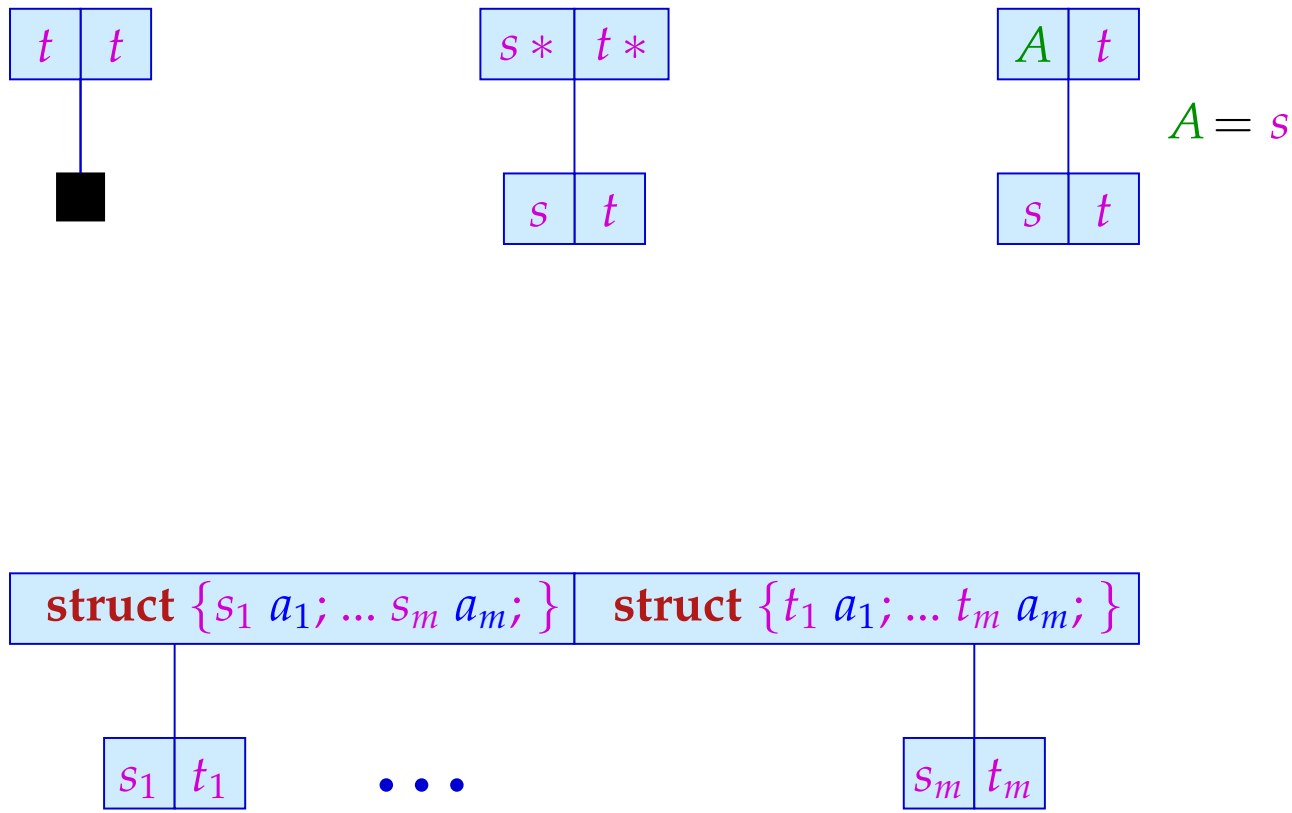
Idee:

- 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

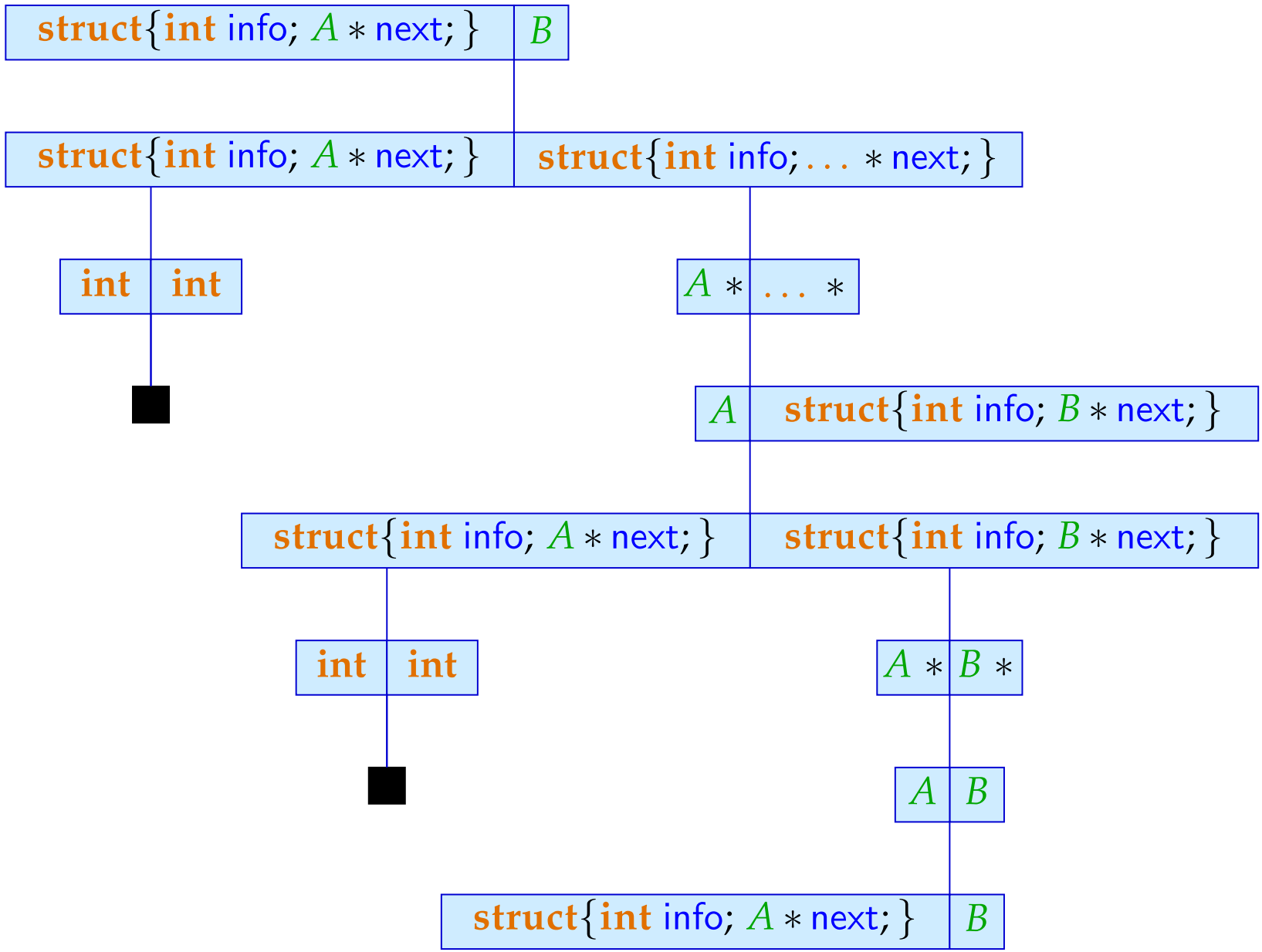
... im Beispiel:

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

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 ...

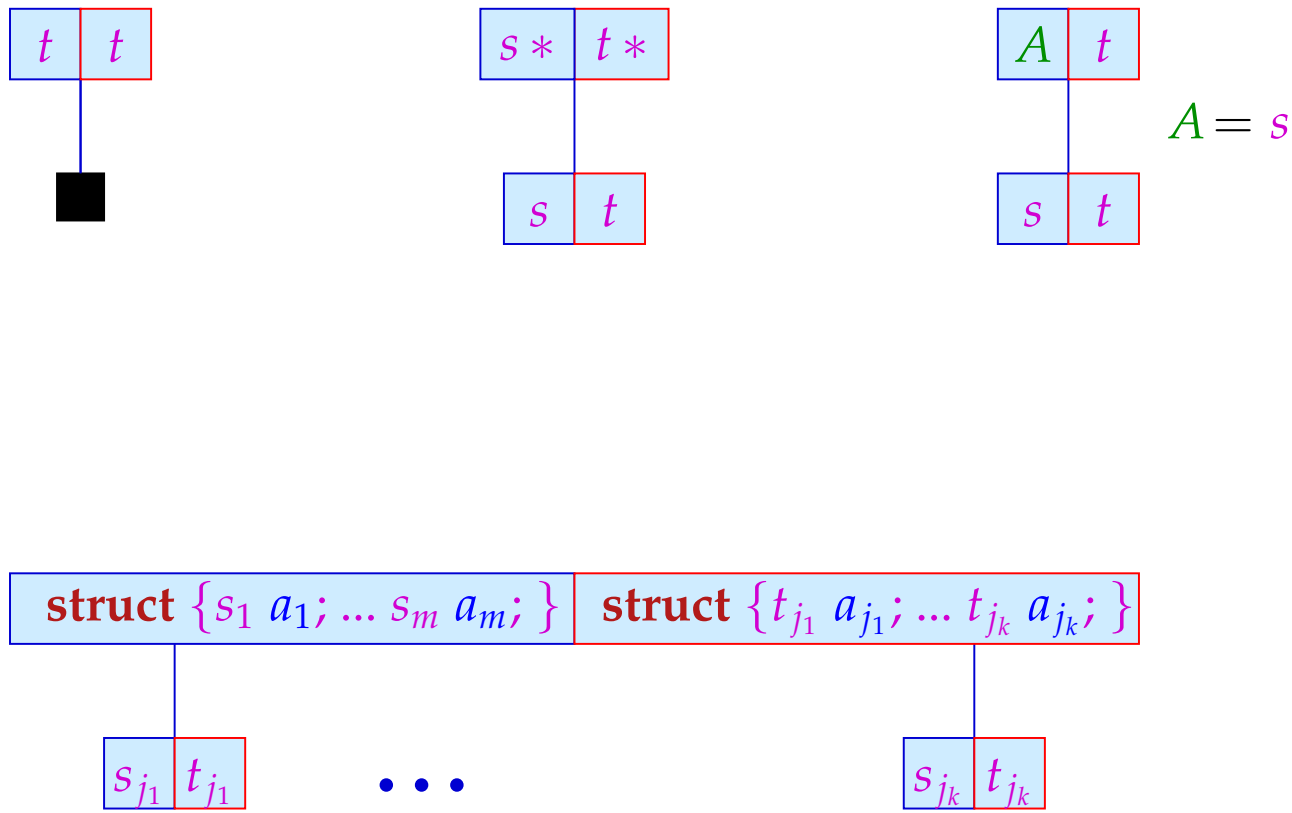
$\Longrightarrow$

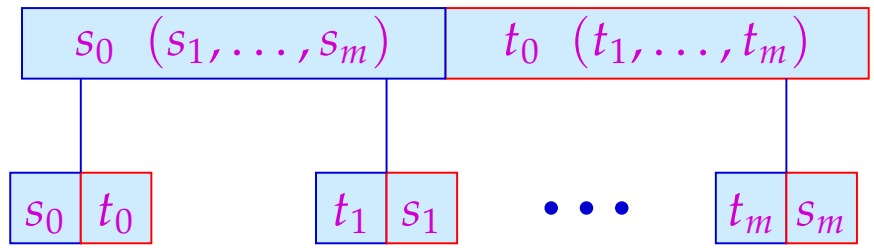
Erweitere Teiltypen-Beziehungen der Basistypen auf komplexe Typen :-)

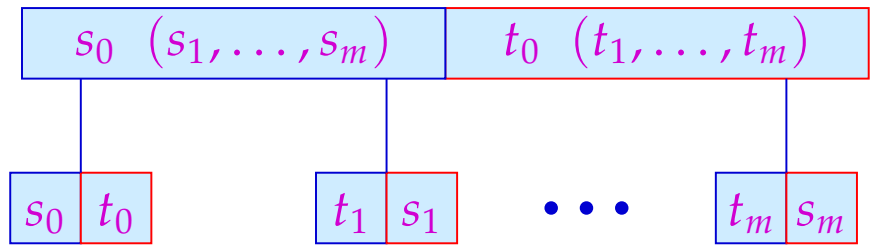
Beispiel:

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

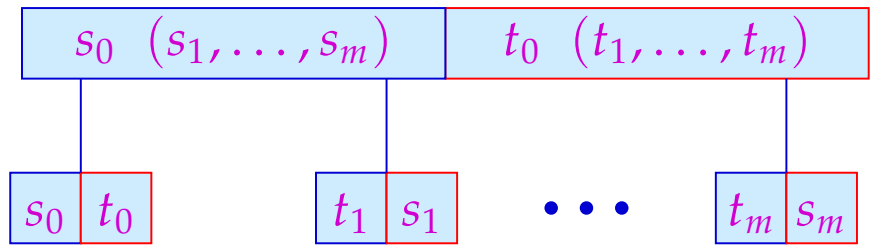

$s_0 \ (s_1, \ldots, s_m)$
$t_0 \ (t_1, \ldots, t_m)$
s_0
t_0
t_1
s_1
t_m
s_m



Beispiele:

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

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



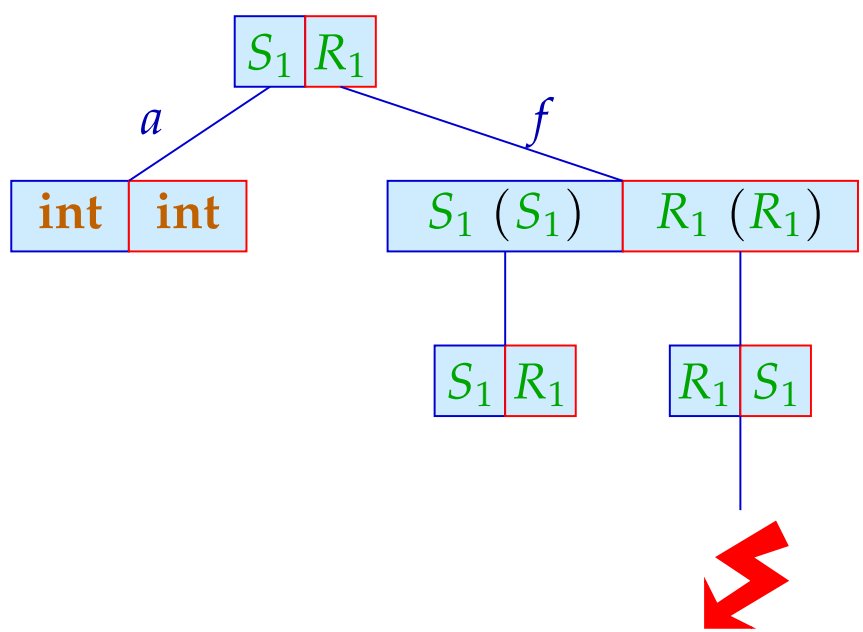
Beispiele:

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

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

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:

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