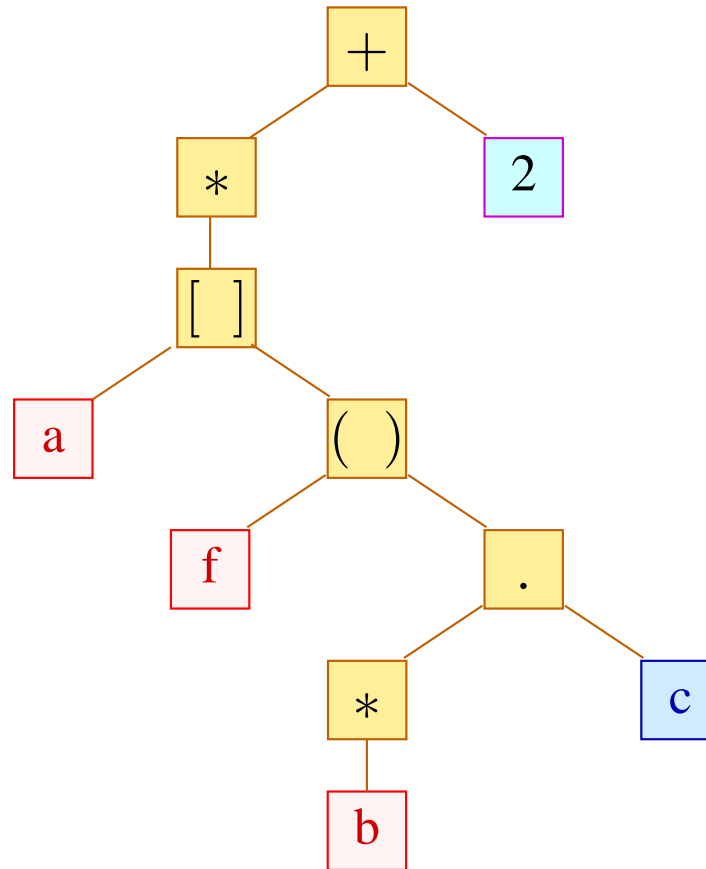
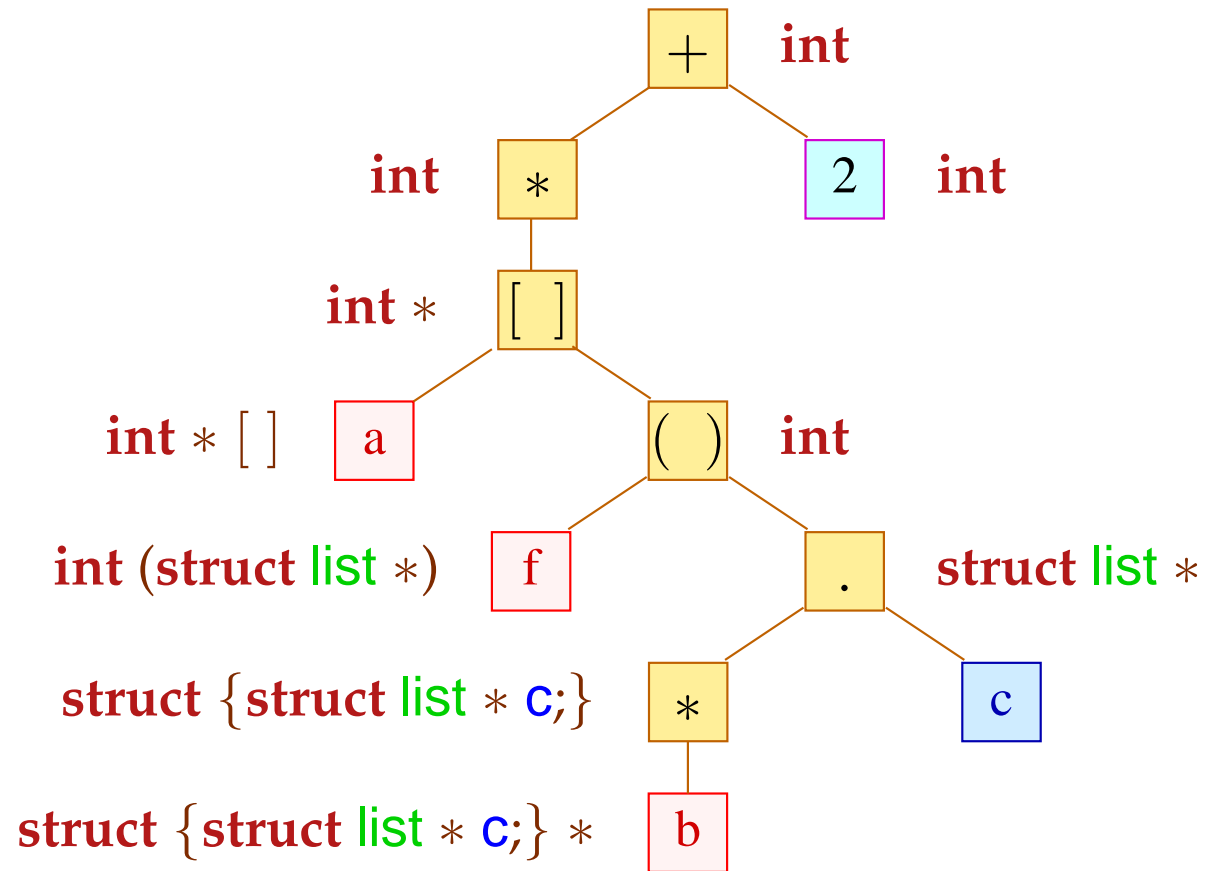


... im Beispiel:



... im Beispiel:



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.

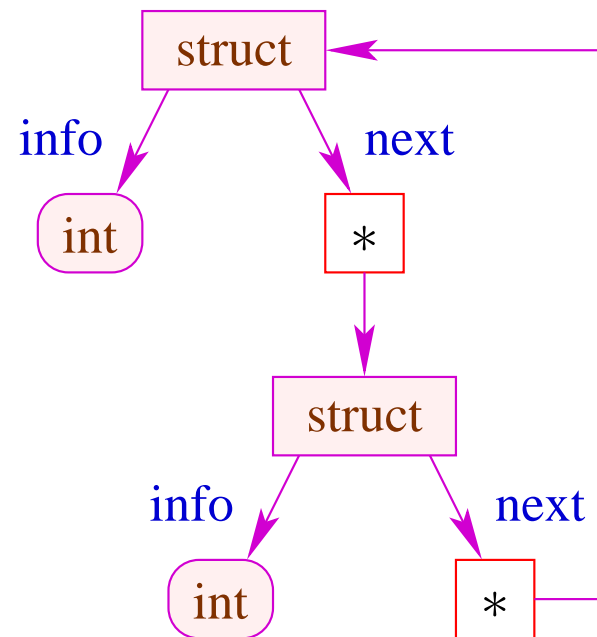
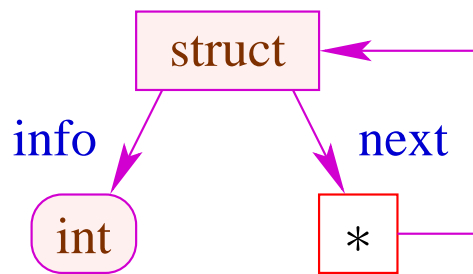
Beispiel:

```
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 **DFA**s äquivalent sind.
- Insbesondere gibt es stets einen eindeutig bestimmten **minimalen** Typgraphen für jeden Typ :-)
- Strukturelle Äquivalenz rekursiver Typen ist deshalb schnell entscheidbar
!!!

Alternativer Algorithmus:

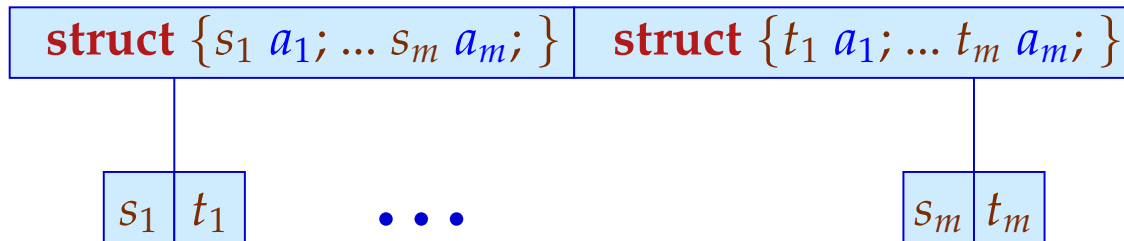
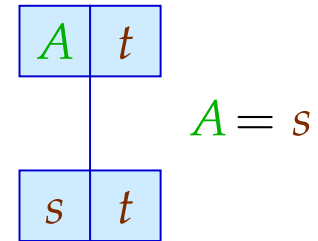
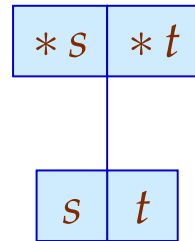
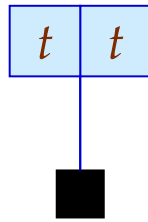
Idee:

- 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 ...



... im Beispiel:

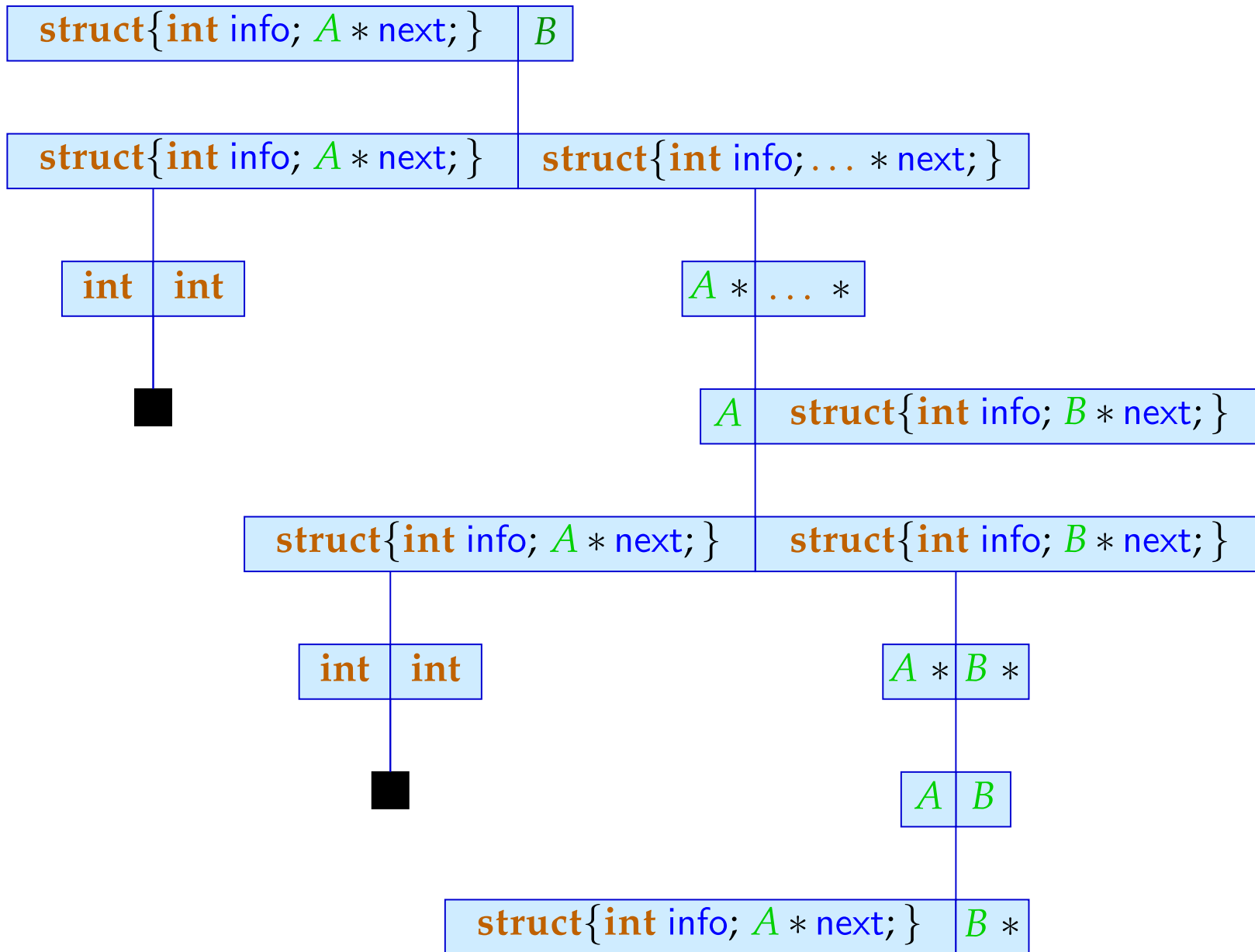
$A = \text{struct } \{\text{int info; } A * \text{next;}\}$

$B = \text{struct } \{\text{int info;}$
 $\text{struct } \{\text{int info; } B * \text{next;}\} * \text{next;}\}$

Wir fragen uns etwa, ob gilt:

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

Dazu konstruieren wir:



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 ;-)

⇒ die Anzahl zu betrachtender Anfragen ist endlich :-)

⇒ 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 ...

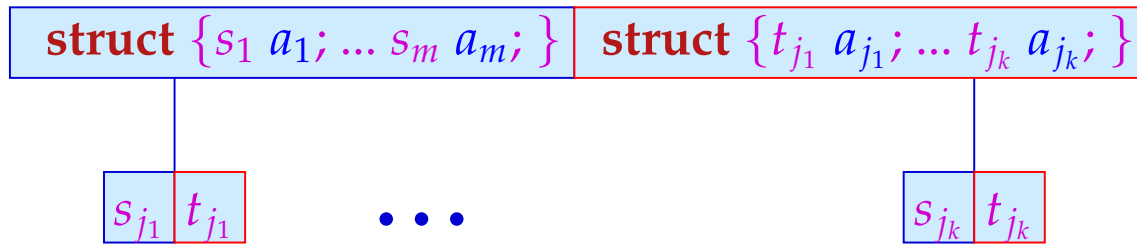
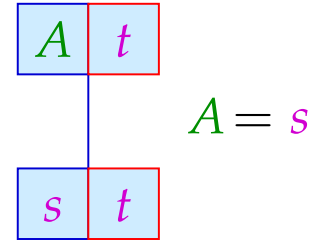
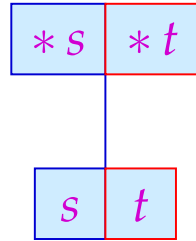
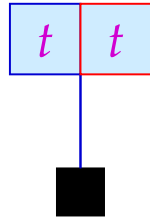


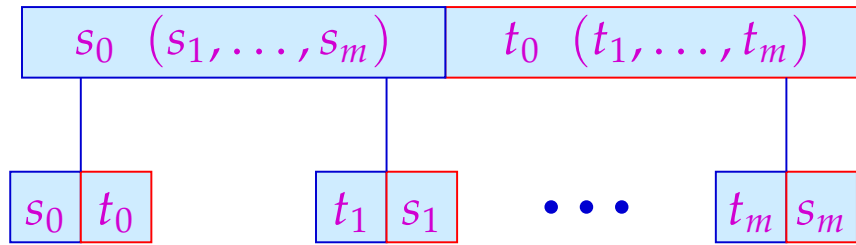
Erweitere Teiltypen-Beziehungen der Basistypen auf komplexe Typen :-)

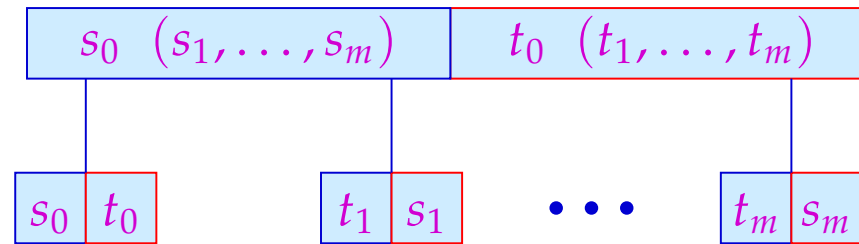
Beispiel:

```
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 ...

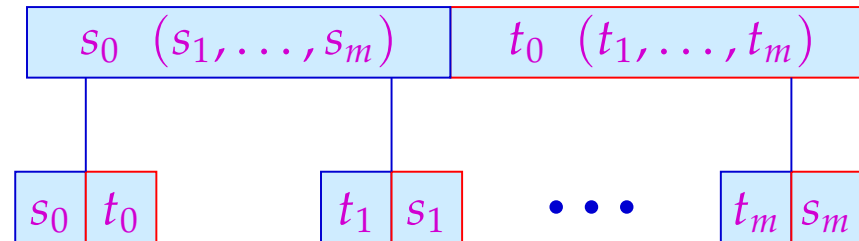






Beispiele:

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



Beispiele:

`struct {int a; int b;}` \leq `struct {float a;}`
`int (int)` $\not\leq$ `float (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 :-)

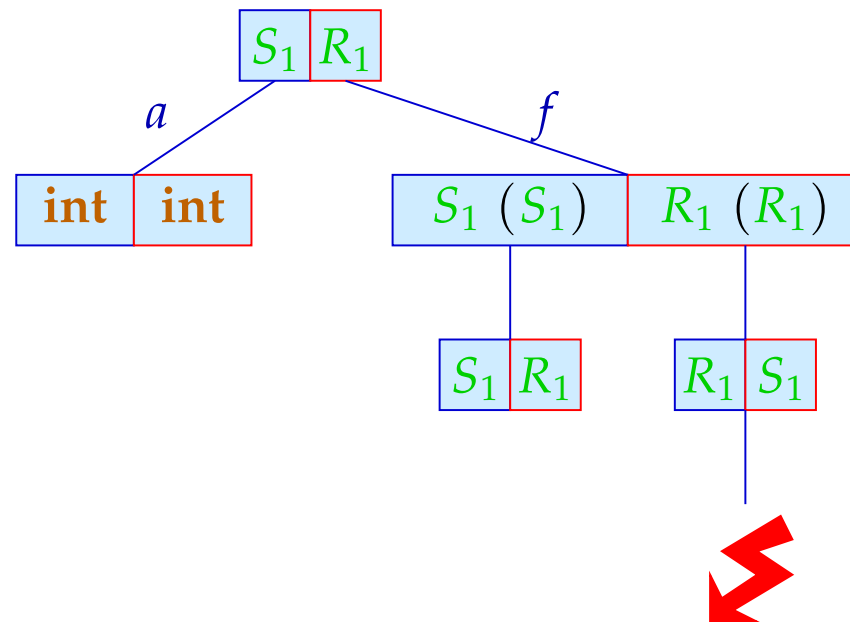
Beispiel:

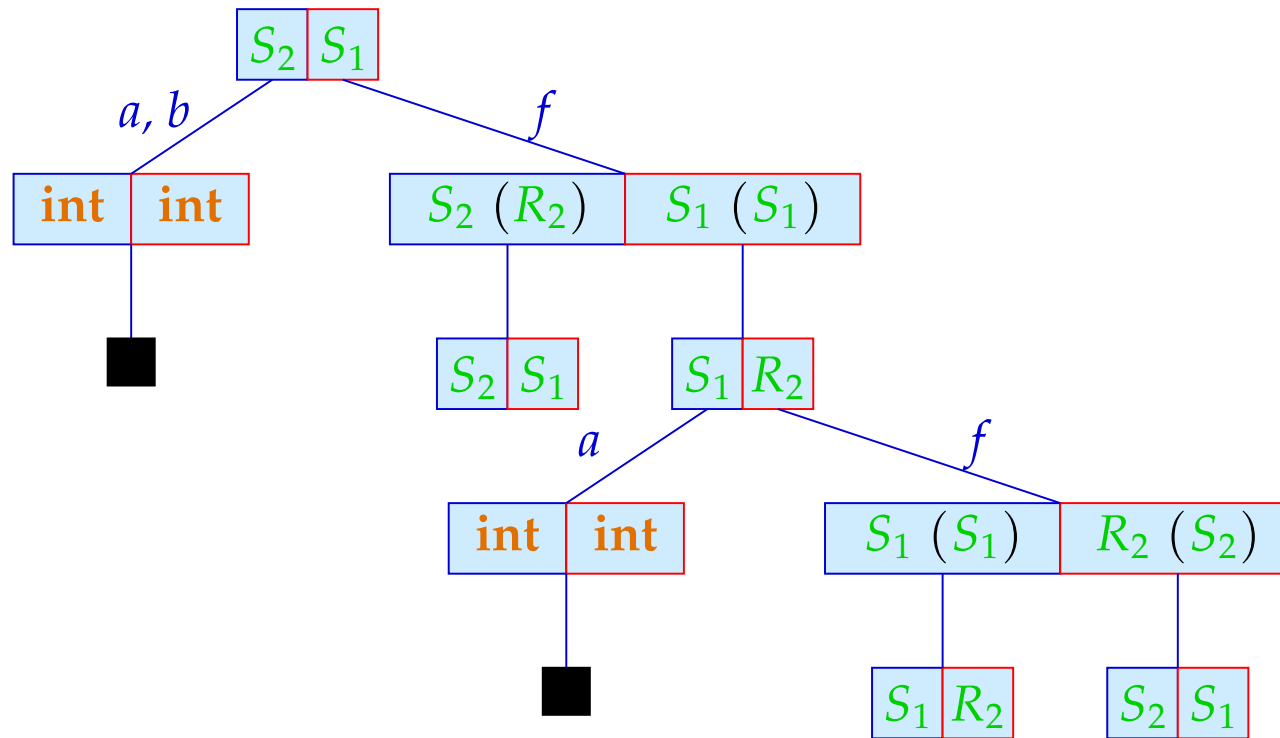
$R_1 = \text{struct } \{\text{int } a; R_1(R_1) f;\}$

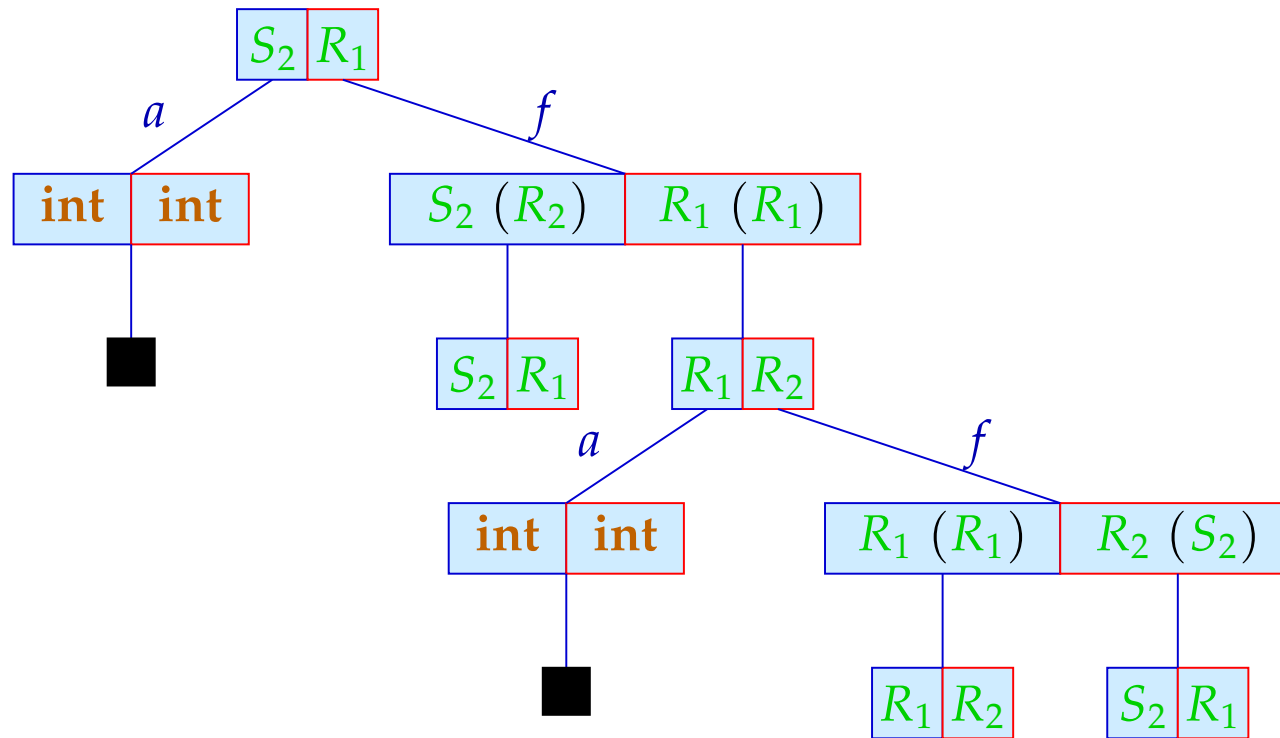
$S_1 = \text{struct } \{\text{int } a; \text{int } b; S_1(S_1) f;\}$

$R_2 = \text{struct } \{\text{int } a; R_2(S_2) f;\}$

$S_2 = \text{struct } \{\text{int } a; \text{int } b; S_2(R_2) f;\}$







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 :-)

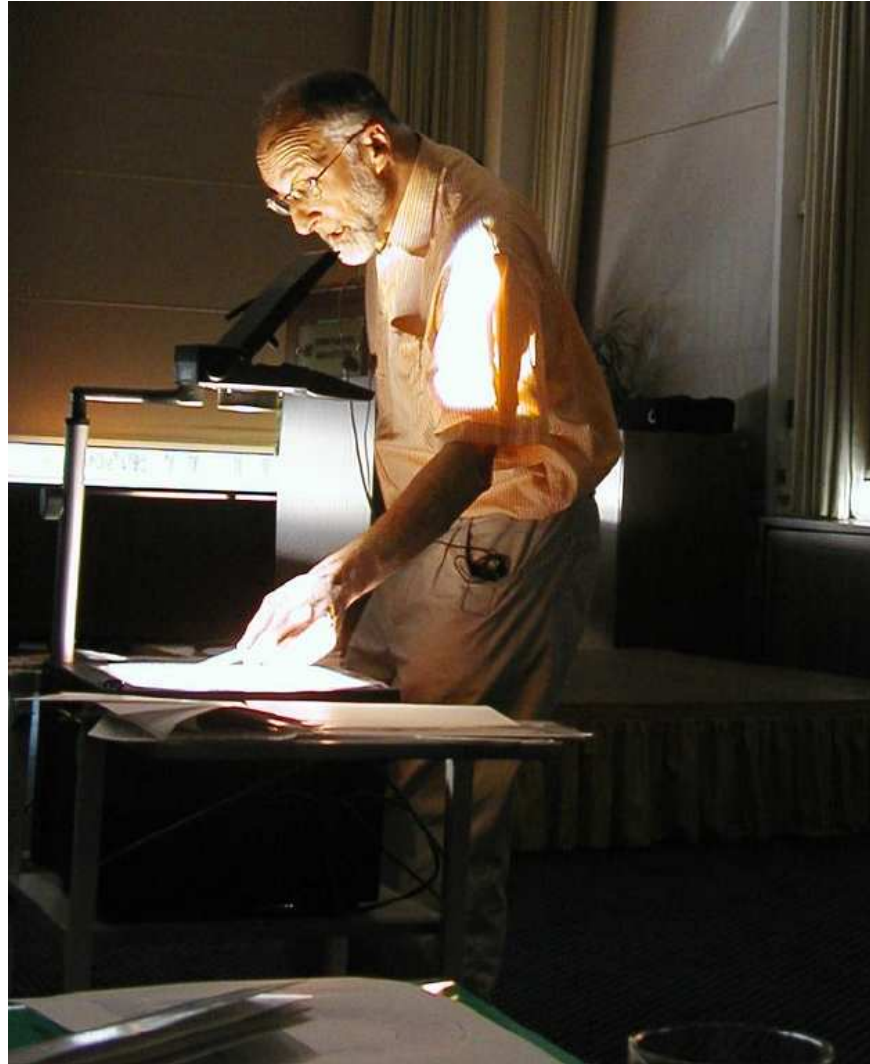
3.3 Inferieren von Typen

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

Beispiel:

```
fun fac x = if x ≤ 0 then 1
           else x · fac (x - 1)
```

Dafür findet der **SML**-Compiler: **fac : int → int**



Robin (Dumbledore) Milner, Edinburgh