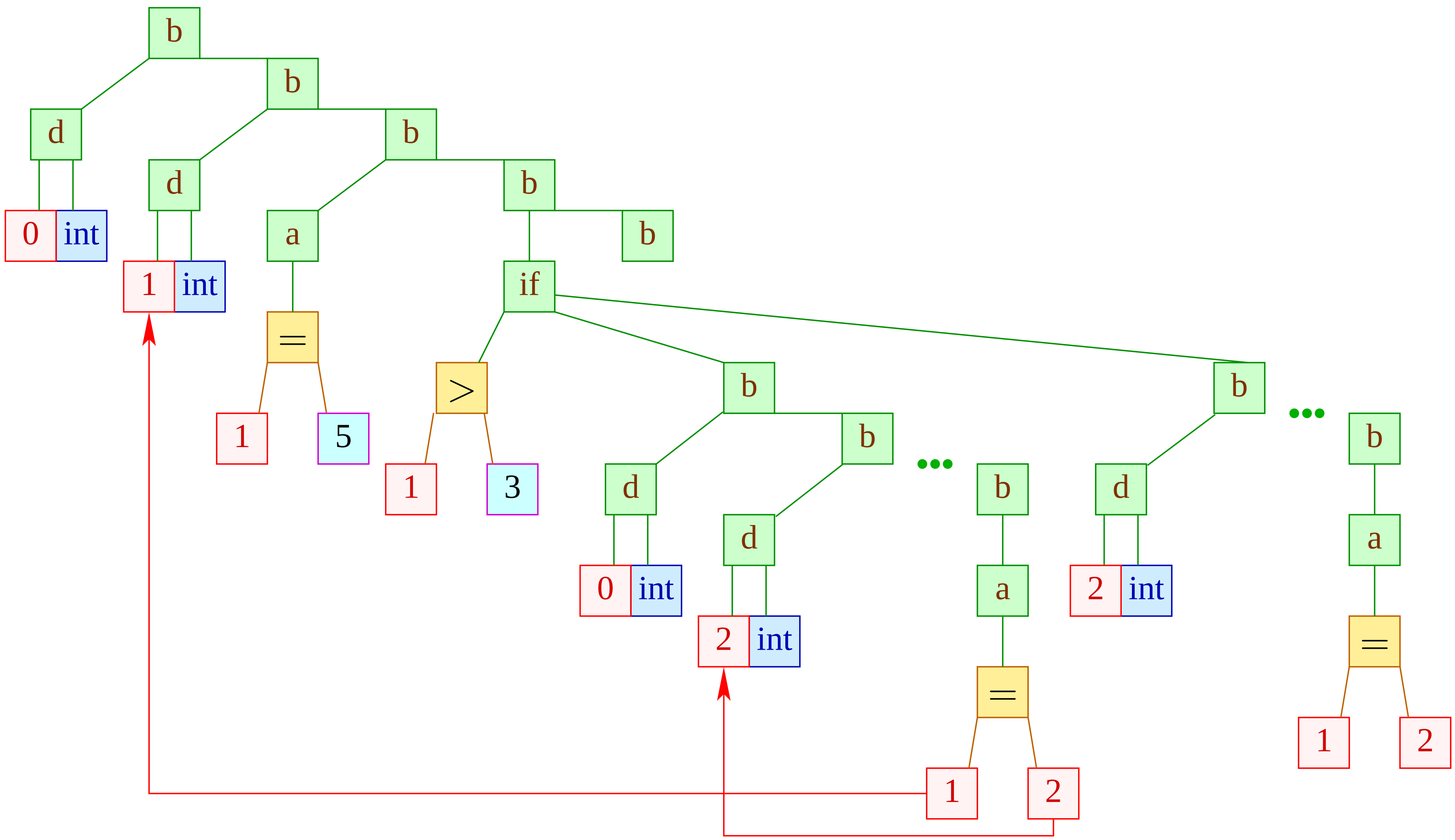
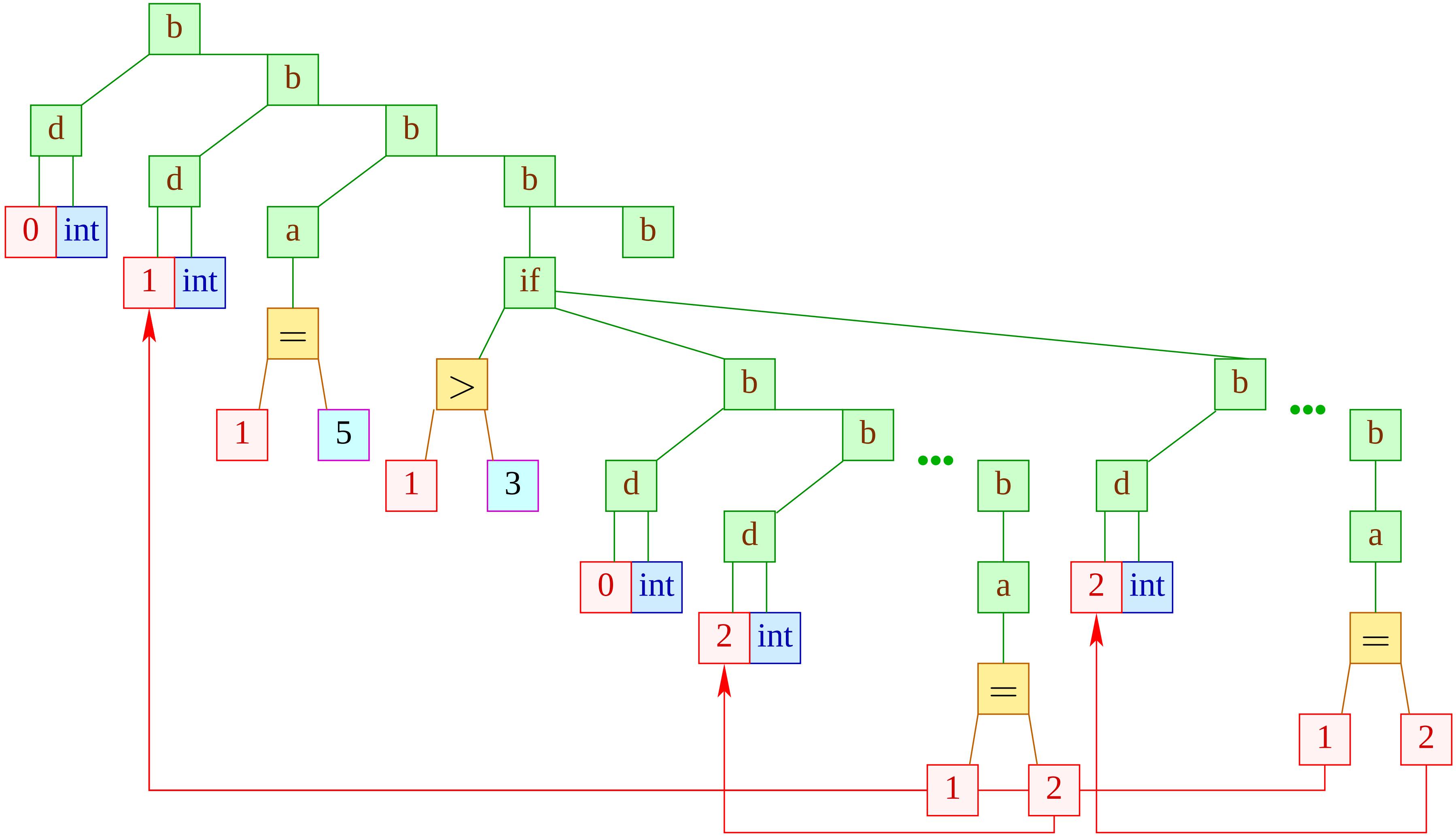

b
b
d
b
0
int
d
b
1
int
a
b
if
b
=
1
5
>
1
3
b
b
d
b
d
0
int
2
int
b
a
=
1
2
d
2
int
b
a
=
1
2


b
d
0
int
b
d
1
int
b
a
=
1
5
b
if
>
1
3
b
b
d
0
int
b
d
2
int
b
a
=
1
2
b
d
2
int
b
a
=
1
2

Diskussion:

- Der Durchlauf ist hier einfach links-rechts DFS.
- Benutzt man eine Listen-Implementierung der Keller und eine rekursive Implementierung, kann man auf das Beseitigen der jeweils neuen Definitionen verzichten :-)
- Anstelle erst die Namen durch Nummern zu ersetzen und dann die Zuordnung von Benutzungen zu Definitionen vorzunehmen, kann man auch gleich eindeutige Nummern vergeben :-))

Diskussion:

- Der Durchlauf ist hier einfach links-rechts DFS.
- Benutzt man eine Listen-Implementierung der Keller und eine rekursive Implementierung, kann man auf das Beseitigen der jeweils neuen Definitionen verzichten :-)
- Anstelle erst die Namen durch Nummern zu ersetzen und dann die Zuordnung von Benutzungen zu Definitionen vorzunehmen, kann man auch gleich eindeutige Nummern vergeben :-))

Achtung:

- Manche Programmiersprachen verbieten eine Mehrfach-Deklaration des selben Namens innerhalb eines Blocks ;-)
- Dann muss man für jede Deklaration einen Pointer auf den Block verwalten, zu dem sie gehört.
- Gibt es eine weitere Deklaration des gleichen Namens mit dem selben Pointer, muss ein Fehler gemeldet werden :-))

Erweiterung:

- Hat man mehrere wechselseitig rekursive Funktionsdefinitionen in einem Block, müssen deren Namen vor Durchmustern der Rümpfe in die Tabelle eingetragen werden ...

$$
\begin{array}{llcl}
\textbf{fun} & \mathsf{odd}\ 0 & = & \textbf{false} \\
| & \mathsf{odd}\ 1 & = & \textbf{true} \\
| & \mathsf{odd}\ x & = & \mathsf{even}\ (x-1) \\
\textbf{and} & \mathsf{even}\ 0 & = & \textbf{true} \\
| & \mathsf{even}\ 1 & = & \textbf{false} \\
| & \mathsf{even}\ x & = & \mathsf{odd}\ (x-1)
\end{array}
$$

- Hat man eine objektorientierte Sprache mit Vererbung zwischen Klassen, sollte die übergeordnete Klasse vor der Unterklasse besucht werden :-)
- Bei Überladung muss simultan eine Typüberprüfung vorgenommen werden ...

3.2 Typ-Überprüfung

In modernen (imperativen / objektorientierten / funktionalen) Programmiersprachen besitzen Variablen und Funktionen einen Typ, z.B. **int**, **struct** { **int** x; **int** y; }.

Typen sind nützlich für:

- die Speicherverwaltung;
- die Vermeidung von Laufzeit-Fehlern :-)

In imperativen /objektorientierten Programmiersprachen muss der Typ bei der Deklaration spezifiziert und vom Compiler die typ-korrekte Verwendung überprüft werden :-)

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:

  $$\textbf{typedef}\ t\ \ x\ ;$$

- 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 $\Gamma = \{t_1\ x_1; \ldots t_m\ x_m;\}$

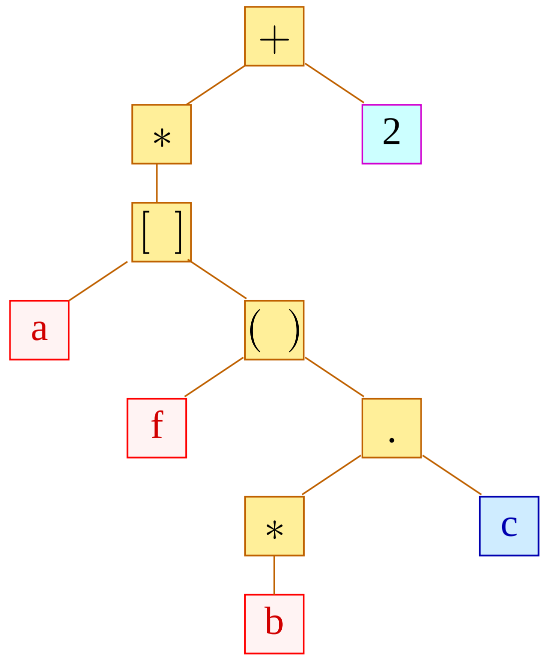
Überprüfe: Kann ein Ausdruck e mit dem Typ t versehen werden?

Beispiel:

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

Betrachte den Ausdruck:

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



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

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

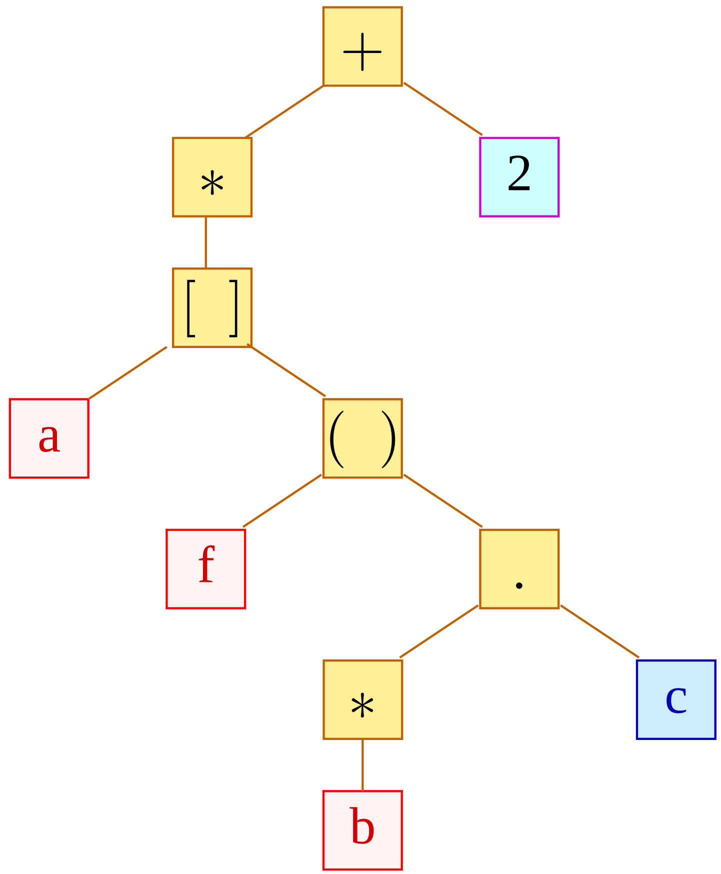
Struct: $$\frac{\Gamma \vdash e : \mathbf{struct}\ \{t_1\ a_1; \ldots t_m\ a_m;\}}{\Gamma \vdash e.a_i : t_i}$$

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

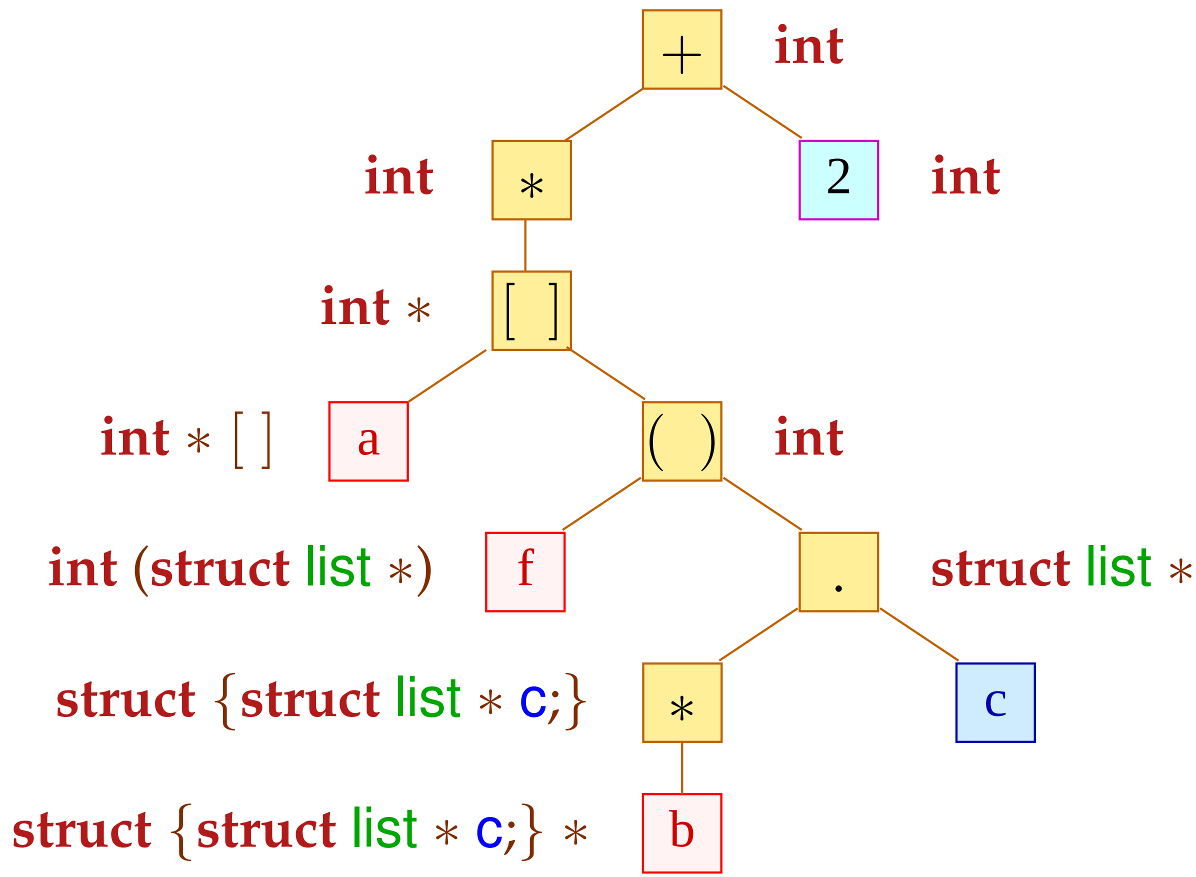
Op: $$\frac{\Gamma \vdash e_1 : \mathbf{int} \qquad \Gamma \vdash e_2 : \mathbf{int}}{\Gamma \vdash e_1 + e_2 : \mathbf{int}}$$

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:

  $\longrightarrow$ 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.

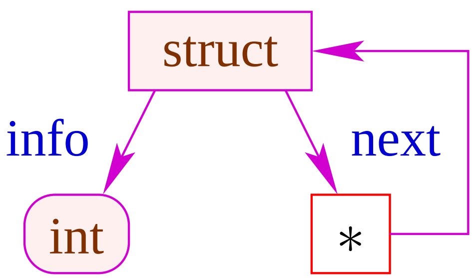
Beispiel:

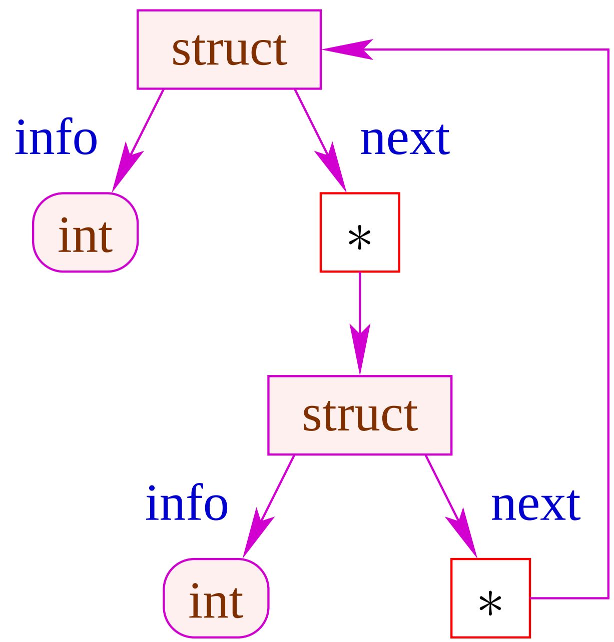
```
struct list {
    int info;
    struct list * next;
}
```

```
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:

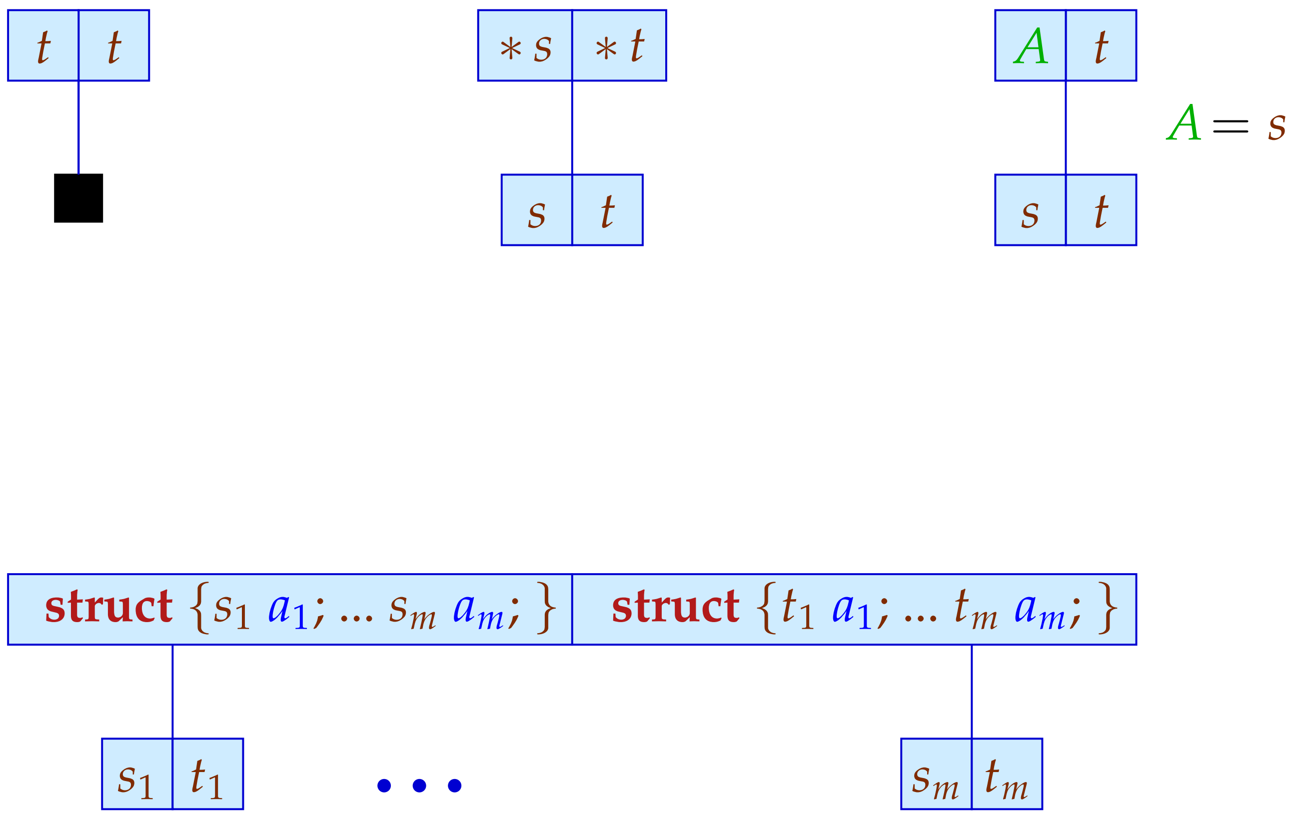
Idee:

- 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

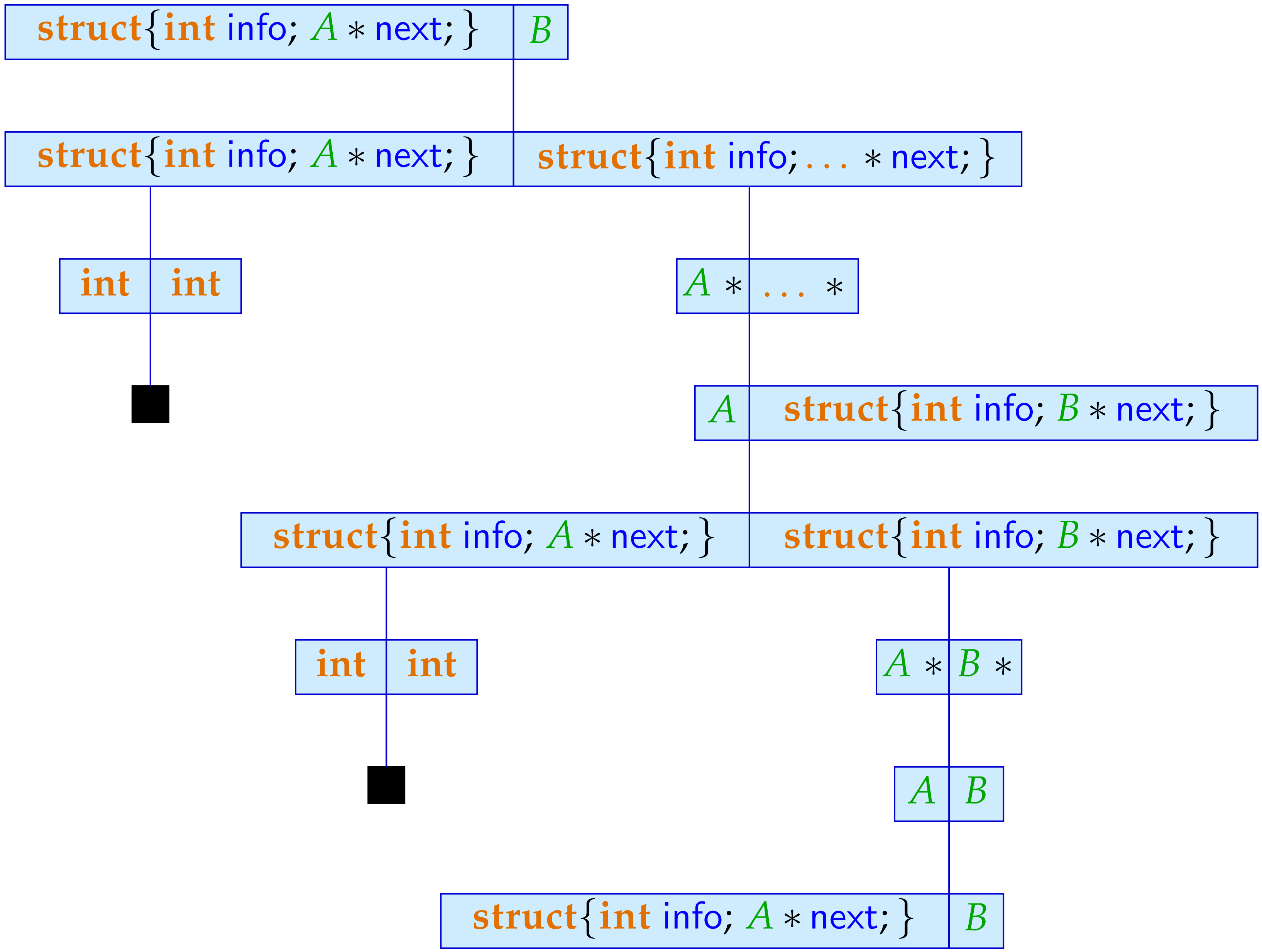
... im Beispiel:

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

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 :-))