Script generated by TTT

Title: Petter: Compilerbau (27.06.2016)

Date: Mon Jun 27 14:26:02 CEST 2016

Duration: 100:03 min

Pages: 40

Type Definitions in C

A type definition is a *synonym* for a type expression.
In C they are introduced using the **typedef** keyword.
Type definitions are useful

- as abbreviation:

```
typedef struct { int x; int y; } point_t;
```

- to construct *recursive* types:

Possible declaration in C:

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

more readable:

```
typedef struct list list_t;
struct list {
  int info;
  list_t* next;
}
list_t* head;
```


Type Definitions in C

The C grammar distinguishes `typedef-name` and `identifier`.
Consider the following declarations:

```
typedef struct { int x,y } point_t;
point_t origin;
```

Typename

Relevant C grammar:

declaration	→	(declaration-specifier)$^+$ declarator ;
declaration-specifier	→	`static` | `volatile` ... `typedef`
		| `void` | `char` | `char` ... `typename`
declarator	→	`identifier` | ...

Type Definitions in C

The C grammar distinguishes `typedef-name` and `identifier`.
Consider the following declarations:

```
typedef struct { int x,y } point_t;
point_t origin;
```

Relevant C grammar:

declaration	→	(declaration-specifier)$^+$ declarator ;
declaration-specifier	→	`static` | `volatile` ... `typedef`
		| `void` | `char` | `char` ... `typename`
declarator	→	`identifier` | ...

Problem:

- parser adds `point_t` to the table of types when the declaration is reduced
- parser state has at least one look-ahead token

Type Definitions in C

The C grammar distinguishes `typedef-name` and `identifier`.
Consider the following declarations:

```
typedef struct { int x,y } point_t;
point_t origin;
```

Relevant C grammar:

declaration	$\to$	(declaration-specifier)$^+$ declarator ;
declaration-specifier	$\to$	`static` | `volatile` ... `typedef`
		| `void` | `char` | `char` ... `typename`
declarator	$\to$	`identifier` | ...

Problem:

- parser adds `point_t` to the table of types when the declaration is reduced
- parser state has at least one look-ahead token
- the scanner has already read `point_t` in line two as identifier

Type Definitions in C: Solutions

Relevant C grammar:

declaration	$\to$	(declaration-specifier)$^+$ declarator ;
declaration-specifier	$\to$	`static` | `volatile` $\cdots$ `typedef`
		| `void` | `char` | `char` $\cdots$ `typename`
declarator	$\to$	`identifier` | $\cdots$

Solution is difficult:

- try to fix the look-ahead inside the parser

Type Definitions in C: Solutions

Relevant C grammar:

declaration	$\to$	(declaration-specifier)$^+$ declarator ;
declaration-specifier	$\to$	`static` | `volatile` $\cdots$ `typedef`
		| `void` | `char` | `char` $\cdots$ `typename`
declarator	$\to$	`identifier` | $\cdots$

Solution is difficult:

- try to fix the look-ahead inside the parser
- add a rule to the grammar: S/R- & R/R- Conflicts!!
  `typename` $\to$ `identifier`

Type Definitions in C: Solutions

Relevant C grammar:

declaration	$\to$	(declaration-specifier)$^+$ declarator ;
declaration-specifier	$\to$	`static` | `volatile` $\cdots$ `typedef`
		| `void` | `char` | `char` $\cdots$ `typename`
declarator	$\to$	`identifier` | $\cdots$

Solution is difficult:

- try to fix the look-ahead inside the parser
- add a rule to the grammar: S/R- & R/R- Conflicts!!
  `typename` $\to$ `identifier`
- register type name earlier

Chapter 3:

Type Checking

Goal of Type Checking

In most mainstream (imperative / object oriented / functional) programming languages, variables and functions have a fixed type. for example: `int`, `void*`, `struct { int x; int y; }`.

Goal of Type Checking

In most mainstream (imperative / object oriented / functional) programming languages, variables and functions have a fixed type. for example: `int`, `void*`, `struct { int x; int y; }`.

Types are useful to

- manage memory
- to avoid certain run-time errors

Goal of Type Checking

In most mainstream (imperative / object oriented / functional) programming languages, variables and functions have a fixed type. for example: `int`, `void*`, `struct { int x; int y; }`.

Types are useful to

- manage memory
- to avoid certain run-time errors

In imperative and object-oriented programming languages a declaration has to specify a type. The compiler then checks for a type correct use of the declared entity.

Type Expressions

Types are given using type-*expressions*.
The set of type expressions T contains:

1. base types: **int**, **char**, **float**, **void**, ...
2. type constructors that can be applied to other types

Type Expressions

Types are given using type-*expressions*.
The set of type expressions T contains:

1. base types: **int**, **char**, **float**, **void**, ...
2. type constructors that can be applied to other types

example for type constructors in C:

- structures: **struct** $\{ t_1\ a_1; \ldots t_k\ a_k; \}$
- pointers: $t\ *$
- arrays: $t\ []$
  - the size of an array can be specified
  - the variable to be declared is written between t and $[n]$
- functions: $t\ (t_1, \ldots, t_k)$
  - the variable to be declared is written between t and $(t_1, \ldots, t_k)$
  - in ML function types are written as: $t_1 * \ldots * t_k \rightarrow t$

Type Checking

Problem:

Given: A set of type declarations $\Gamma = \{t_1\ x_1; \ldots t_m\ x_m;\}$
Check: Can an expression e be given the type t?

Type Checking

Problem:

Given: A set of type declarations $\Gamma = \{t_1\ x_1; \ldots t_m\ x_m;\}$
Check: Can an expression e be given the type t?

Example:

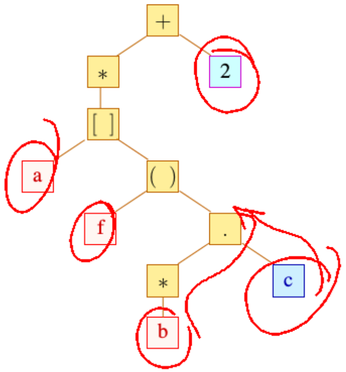
```
struct list { int info; struct list* next; };
int f(struct list* l) { return 1; };
struct { struct list* c;}* b;
int* a[11];
```

Consider the expression:

```
*a[f(b->c)]+2;
```

Type Checking using the Syntax Tree

Check the expression `*a[f(b->c)]+2`:



Idea:

- traverse the syntax tree bottom-up
- for each identifier, we lookup its type in Γ
- constants such as 2 or 0.5 have a fixed type
- the types of the inner nodes of the tree are deduced using *typing rules*

Type Systems

Formally: consider *judgements* of the form:

$$\Gamma \vdash e \; : \; t$$

// (in the type environment Γ the expression e has type t)

Axioms:

Const: $\Gamma \vdash c \; : \; t_c$ (t_c type of constant c)

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

Rules:

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

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

Type Systems for C-like Languages

More rules for typing an expression:

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

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

Explicit Cast: $$\frac{\Gamma \vdash e \; : \; t_1 \qquad t_1 \text{ can be converted to } t_2}{\Gamma \vdash (t_2) \; e \; : \; t_2}$$

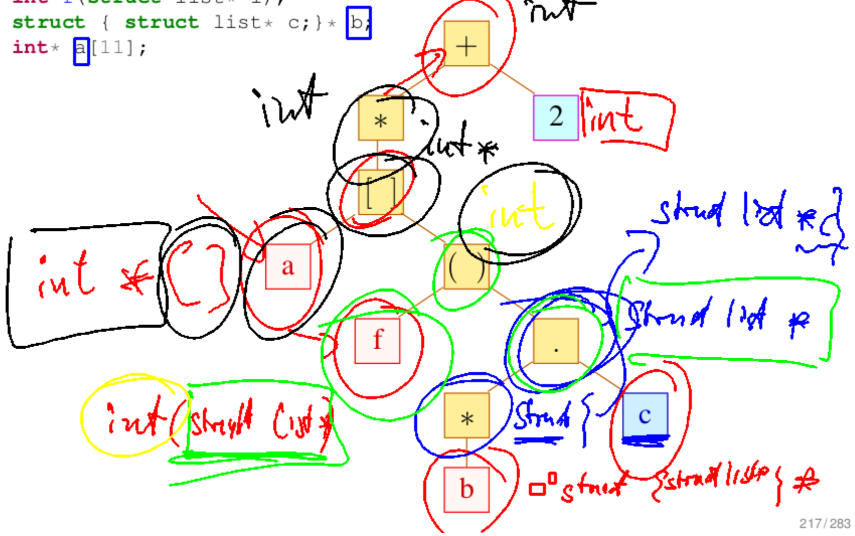
Example: Type Checking

Given expression `*a[f(b->c)]+2` and

Γ = {

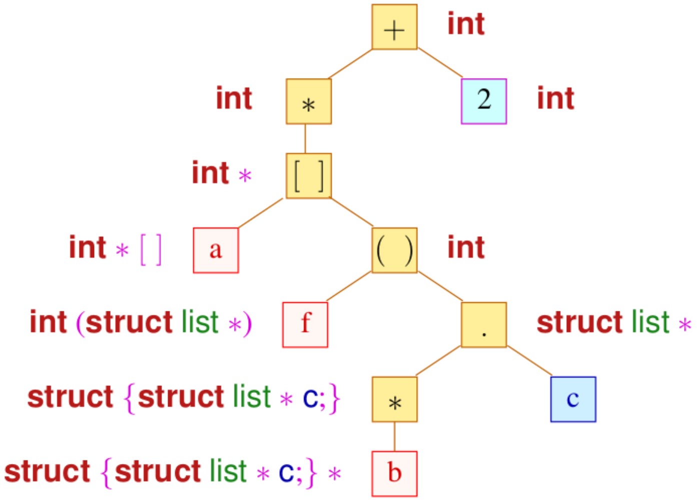
```
  struct list { int info; struct list* next; };
  int f(struct list* l);
  struct { struct list* c;}* b;
  int* a[11];
```

}



Example: Type Checking

Given expression `*a[f(b->c)]+2`:



Equality of Types

Summary of Type Checking

- Choosing which rule to apply at an AST node is determined by the type of the child nodes
- determining the rule requires a check for ⇝ *equality* of types

type equality in C:

- `struct A {}` and `struct B {}` are considered to be different
  - ⇝ the compiler could re-order the fields of `A` and `B` independently (*not* allowed in C)
  - to extend an record `A` with more fields, it has to be embedded into another record:

```
struct B {
   struct A;
   int field_of_B;
} extension_of_A;
```

- after issuing `typedef int C;` the types `C` and `int` are the same

Structural Type Equality

Alternative interpretation of type equality (*does not hold in C*):

semantically, two types t_1, t_2 can be considered as *equal* if they accept the same set of access paths.

Example:

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

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

Consider declarations `struct list* l` and `struct list1* l`. Both allow

`l->info   l->next->info`

but the two declarations of `l` have unequal types in C.

Algorithm for Testing Structural Equality

Idea:

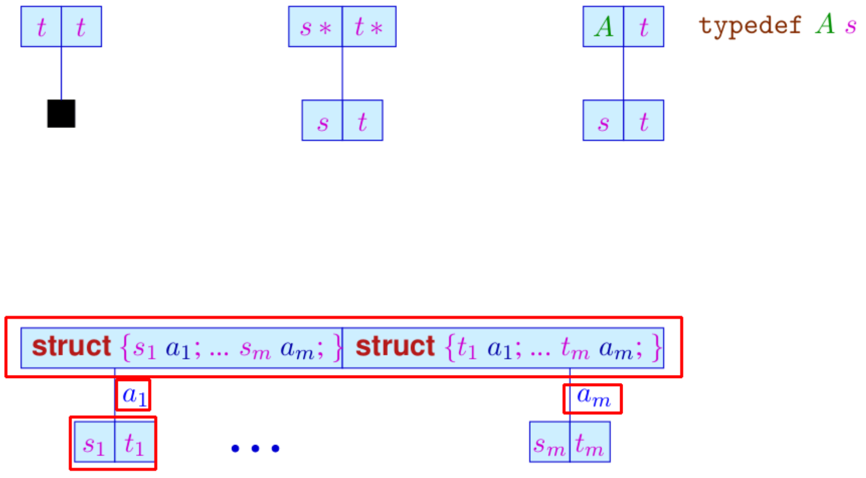
- track a set of equivalence queries of type expressions
- if two types are syntactically equal, we stop and report success
- otherwise, reduce the equivalence query to a several equivalence queries on (hopefully) simpler type expressions

Suppose that recursive types were introduced using type definitions:

typedef A t

(we omit the Γ). Then define the following rules:

Rules for Well-Typedness



Example:

$$\begin{array}{lll} \texttt{typedef} & \texttt{struct}\ \{\texttt{int}\ \ \text{info};\ \ A * \text{next};\ \} & A \\ \texttt{typedef} & \texttt{struct}\ \{\texttt{int}\ \ \text{info};\ \ \texttt{struct}\ \{\texttt{int}\ \ \text{info};\ \ B * \text{next};\ \} * \text{next};\ \} & B \end{array}$$

We ask, for instance, if the following equality holds:

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

We construct the following deduction tree:

Example:

$$\begin{array}{lll} \texttt{typedef} & \texttt{struct}\ \{\texttt{int}\ \ \text{info};\ \ A * \text{next};\ \} & A \\ \texttt{typedef} & \texttt{struct}\ \{\texttt{int}\ \ \text{info};\ \ \texttt{struct}\ \{\texttt{int}\ \ \text{info};\ \ B * \text{next};\ \} * \text{next};\ \} & B \end{array}$$

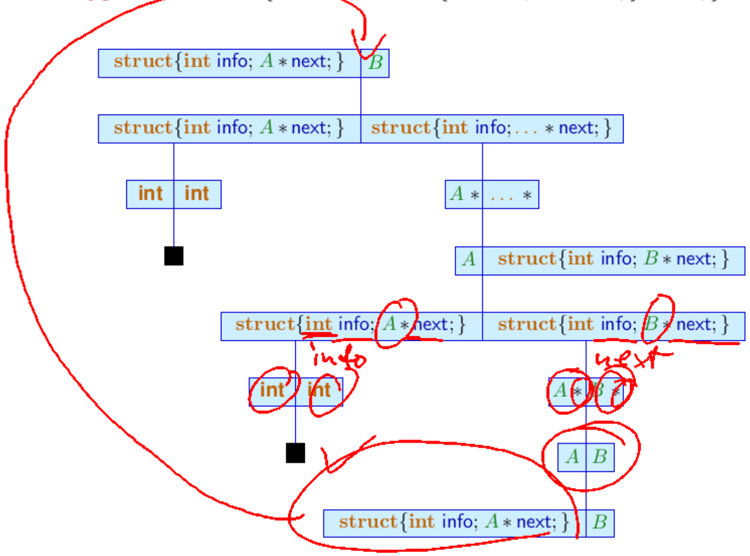
We ask, for instance, if the following equality holds:

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

We construct the following deduction tree:

Proof for the Example:

$$\begin{array}{lll} \texttt{typedef} & \texttt{struct}\ \{\texttt{int}\ \ \text{info};\ \ A * \text{next};\ \} & A \\ \texttt{typedef} & \texttt{struct}\ \{\texttt{int}\ \ \text{info};\ \ \texttt{struct}\ \{\texttt{int}\ \ \text{info};\ \ B * \text{next};\ \} * \text{next};\ \} & B \end{array}$$



Implementation

We implement a function that implements the equivalence query for two types by applying the deduction rules:

- if no deduction rule applies, then the two types are *not equal*
- if the deduction rule for expanding a type definition applies, the function is called recursively with a *potentially larger* type
- in case an equivalence query appears a second time, the types are *equal by definition*

Implementation

We implement a function that implements the equivalence query for two types by applying the deduction rules:

- if no deduction rule applies, then the two types are *not equal*
- if the deduction rule for expanding a type definition applies, the function is called recursively with a *potentially larger* type
- in case an equivalence query appears a second time, the types are *equal by definition*

Termination

- the set D of all declared types is finite
- there are no more than $|D|^2$ different equivalence queries
- repeated queries for the same inputs are automatically satisfied

$\leadsto$ termination is ensured

Overloading and Coercion

Some operators such as $+$ are *overloaded*:

- $+$ has *several possible* types
  for example: `int +(int,int)`, `float +(float, float)`
  but also `float* +(float*, int)`, `int* +(int, int*)`
- depending on the type, the operator $+$ has a different implementation
- determining which implementation should be used is based on the type of the *arguments* only

Overloading and Coercion

Some operators such as $+$ are *overloaded*:

- $+$ has *several possible* types
  for example: `int +(int,int)`, `float +(float, float)`
  but also `float* +(float*, int)`, `int* +(int, int*)`
- depending on the type, the operator $+$ has a different implementation
- determining which implementation should be used is based on the type of the *arguments* only

Coercion: allow the application of $+$ to `int` and `float`.

$\leadsto$ instead of defining $+$ for all possible combinations of types, the arguments are automatically coerced

- conversion is usually done towards more general types i.e. `5+0.5` has type `float` (since `float` $\geq$ `int`)
- coercion may generate code (e.g. converting `int` to `float`)

Subtypes

On the arithmetic basic types **char**, **int**, **long**, etc. there exists a rich *subtype* hierarchy

Subtypes

$t_1 \leq t_2$, means that the values of type t_1

1. form a subset of the values of type t_2;
2. can be converted into a value of type t_2;
3. fulfill the requirements of type t_2;
4. are assignable to variables of type $t2$.

Subtypes

On the arithmetic basic types **char**, **int**, **long**, etc. there exists a rich *subtype* hierarchy

Subtypes

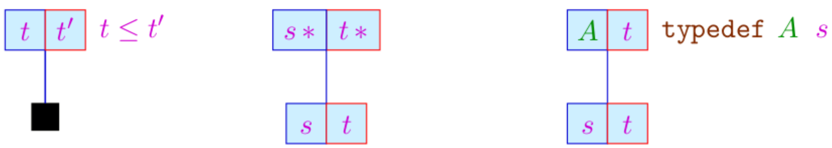
$t_1 \leq t_2$, means that the values of type t_1

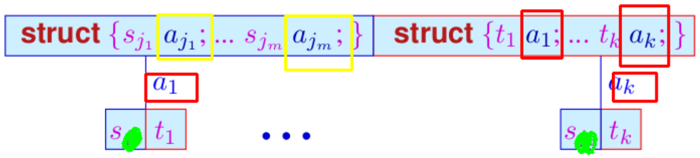
1. form a subset of the values of type t_2;
2. can be converted into a value of type t_2;
3. fulfill the requirements of type t_2;
4. are assignable to variables of type $t2$.

Example:
assign smaller type (fewer values) to larger type (more values)

$$t_1 \quad x;$$
$$t_2 \quad y;$$
$$y = x;$$

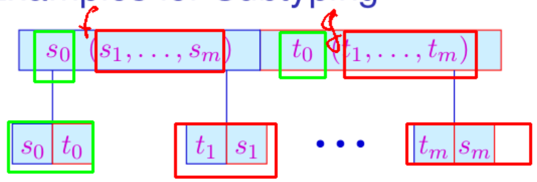
Rules for Well-Typedness of Subtyping





$$\text{struct } \{int\ u, int\ v\} \quad x;$$
$$\text{struct } \{int\ u\} \quad y;$$
$$y = x;$$

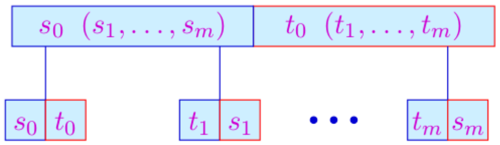
Rules and Examples for Subtyping



Examples:

struct {**int** a; **int** b; }	**struct** {**float** a; }
int (**int**)	**float** (**float**)
int (**float**)	**float** (**int**)

Rules and Examples for Subtyping



Examples:

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

Definition

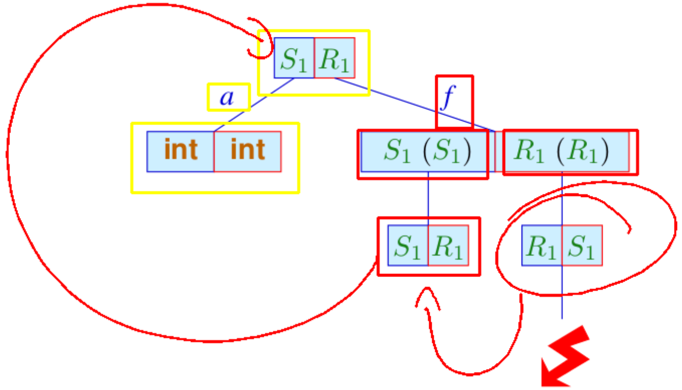
Given two function types in subtype relation $s_0(s_1, \ldots s_n) \leq t_0(t_1, \ldots t_n)$ then we have

- co-variance of the return type $s_0 \leq t_0$ and
- contra-variance of the arguments $s_i \geq t_i$ für $1 < i \leq n$

Subtypes: Application of Rules (I)

Check if $S_1 \leq R_1$:

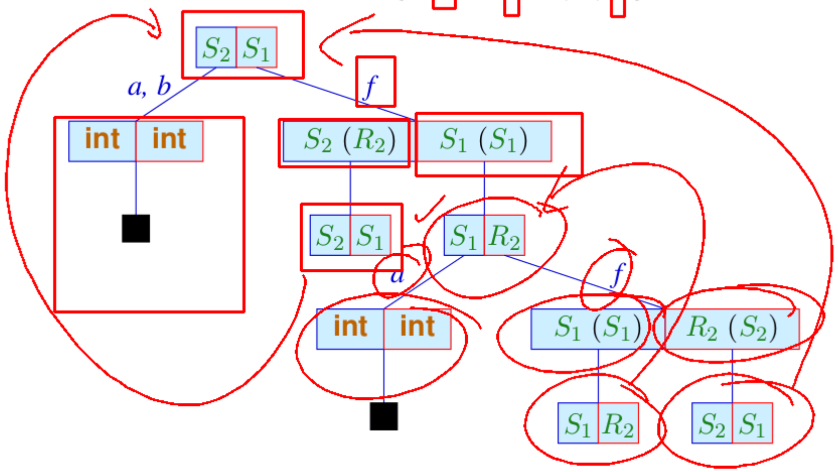
$$\begin{array}{lcl} R_1 & = & \textbf{struct}\ \{\textbf{int}\ a;\ R_1\ (R_1)\ f;\} \\ S_1 & = & \textbf{struct}\ \{\textbf{int}\ a;\ \textbf{int}\ b;\ S_1\ (S_1)\ f;\} \\ R_2 & = & \textbf{struct}\ \{\textbf{int}\ a;\ R_2\ (S_2)\ f;\} \\ S_2 & = & \textbf{struct}\ \{\textbf{int}\ a;\ \textbf{int}\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



Subtypes: Application of Rules (II)

Check if $S_2 \leq S_1$:

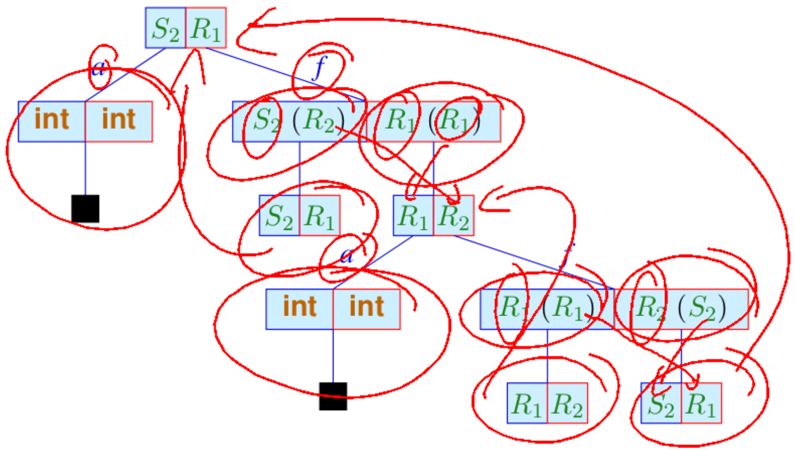
$$\begin{array}{lcl} R_1 & = & \textbf{struct}\ \{\textbf{int}\ a;\ R_1\ (R_1)\ f;\} \\ S_1 & = & \textbf{struct}\ \{\textbf{int}\ a;\ \textbf{int}\ b;\ S_1\ (S_1)\ f;\} \\ R_2 & = & \textbf{struct}\ \{\textbf{int}\ a;\ R_2\ (S_2)\ f;\} \\ S_2 & = & \textbf{struct}\ \{\textbf{int}\ a;\ \textbf{int}\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



Subtypes: Application of Rules (III)

Check if $S_2 \leq R_1$:

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



Discussion

- for presentational purposes, proof trees are often abbreviated by omitting deductions within the tree
- structural sub-types are very powerful and can be quite intricate to understand
- Java generalizes records to objects/classes where a sub-class A inheriting form base class O is a subtype $A \leq O$
- subtype relations between classes must be explicitly declared
- inheritance ensures that all sub-classes contain all (visible) components of the super class
- a shadowed (overwritten) component in A must have a subtype of the the component in O
- Java does not allow argument subtyping for methods since it uses different signatures for overloading