Script generated by TTT

Title:	Simon: Compilerbau (16.06.2014)
Date:	Mon Jun 16 14:16:33 CEST 2014
Duration:	90:22 min
Pages:	29

Chapter 1: 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; }.

int × int

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 $\mathcal{T}$ contains:

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

example for type constructors in C:

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

x : int * int → int
x :: int → (int → int)

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 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:

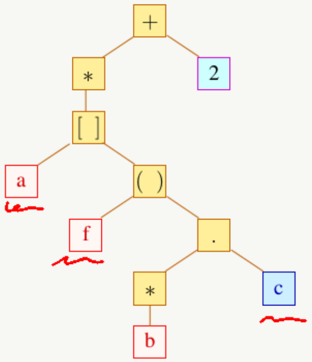
```
struct list { int info; struct list* next; };
int f(struct list* l) { return 1; };
struct { struct list* c;}* b;
int* a[11];
```

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

Formal 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)

Regeln:

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

Type Systems for C-like Languages

More rules for typing an expression:

Op+ $\dfrac{\Gamma \vdash e_1 : t* \qquad \Gamma \vdash e_2 : int}{\Gamma \vdash e_1 + e_2 : t*}$

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

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

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

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

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

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

$\dfrac{\Gamma \vdash e_1 : t* \qquad \Gamma \vdash e_2 : t*}{\Gamma \vdash e_1 - e_2 : t*}$ $\qquad e_1 + (int)\ e_2$

Example: Type Checking

Given expression `*a[f(b->c)]+2` and $\Gamma = \{$

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

struct list *

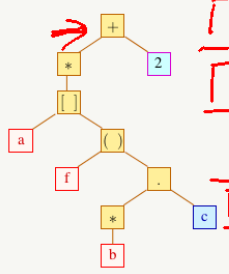
int (struct list *)

int* [11]

$\dfrac{\Gamma \vdash b : \Gamma(b)}{\Gamma \vdash *b : struct\ \{..t\ c\ ..\}*}$ Deref

struct

$\Gamma \vdash (*b)\to c : struct\ list *$



$\dfrac{\Gamma \vdash \Gamma(\ )}{\Gamma \vdash a : \Gamma(a)} \quad \dfrac{\Gamma \vdash f : \Gamma(f) \qquad \Gamma \vdash (*b)\to c : struct\ list*}{\Gamma \vdash f(b\to c) : int}$ App

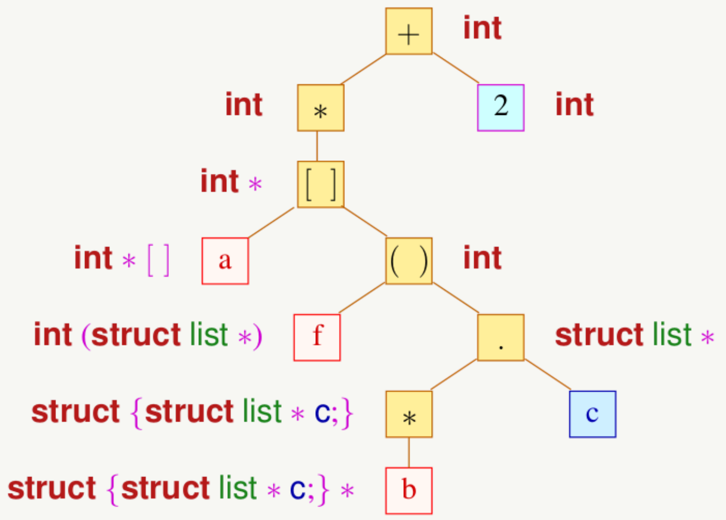
Array $\dfrac{}{\Gamma \vdash a[f(b\to c)] : int*}$

Deref $\dfrac{}{\Gamma \vdash *a[f(b\to c)] : int} \quad \dfrac{}{\Gamma \vdash 2 : int}$ Const

Op $\dfrac{}{\Gamma \vdash *a[f(b\to c)]+2 : int}$

Example: Type Checking

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



Equality of Types

Summary 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:

```
typedef struct B {
          struct A 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 type t_1, t_2 can be considered as *equal* if the 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:

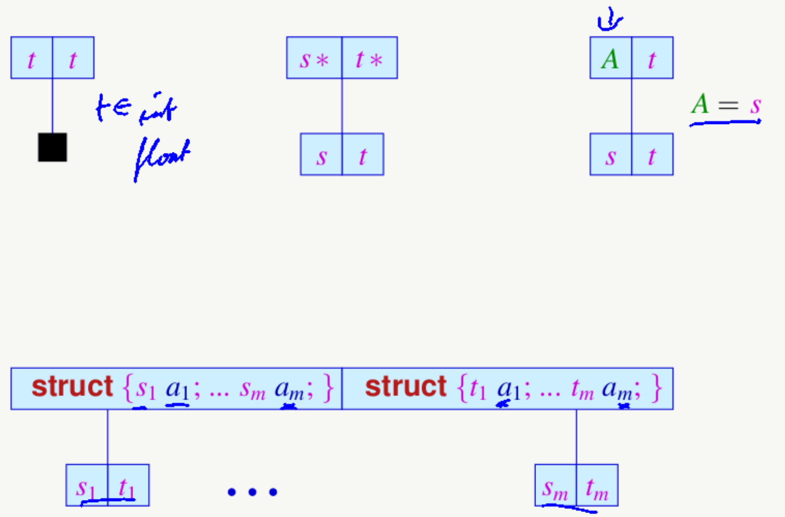
- track a set of equivalence queries of type expressions
- if two types are syntactically equal, we stop and report success
- otherwise, reduce the equivalence query to a several equivalence queries on (hopefully) simpler type expressions

Suppose that recursive types were introduces using type equalities of the form:

$$A = t$$

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

Rules for Well-Typedness



Example:

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

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

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

We construct the following derivation tree:

struct {int info; A* next} = struct {int info, struct {int info, B* next;} * next;}

int, int

A*, struct{ ... }*

A, struct{int info, B* next}

struct{int info; A* next}, struct{int info, B* next}

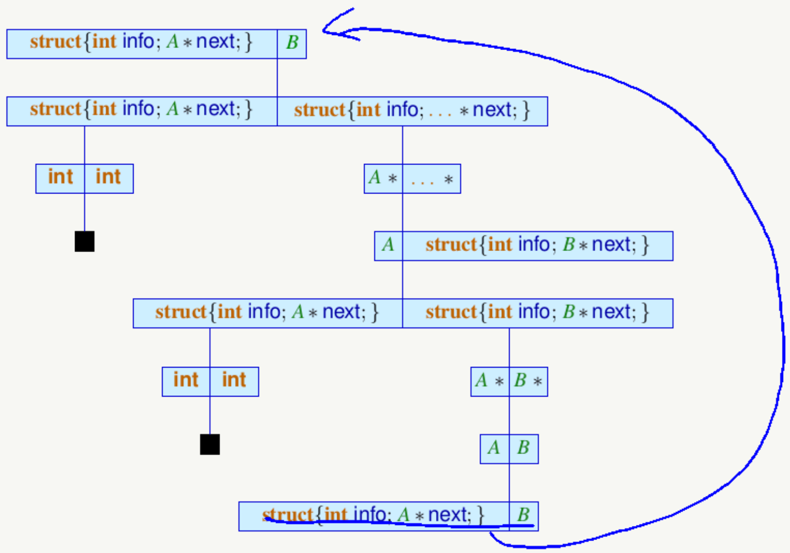
int int

A*, B*

A, B

Proof for the Example:

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



Implementation

We implement a function that implement 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
- during the construction of the proof tree, an equivalence query might occur several times
- in case an equivalence query appears a second time, the types are by definition equal

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 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 *arguments* only

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

- instead of defining $+$ for all possible combinations of types, the arguments are automatically coerced
- this coercion may generate code (z.B. conversion from `int` to `float`)
- coersion is usually done towards more general types i.e. `5+0.5` has type `float` (since `float` $\geq$ `int`)

Coercion of Integer-Types in C: Promotion

C defines special conversion rules for integers: *promotion*

unsigned char / **signed char** $\leq$ **unsigned short** / **signed short** $\leq$ **int** $\leq$ **unsigned int**

. . . where a conversion has to happen via all intermediate types.

subtle errors possible! Compute the character distribution of `str`:

str[1000] = 0;

```
char* str = "...";
int dist[256];
memset(dist, 0, sizeof(dist));
while (*str) {
  dist[(unsigned) *str]++;
  str++;
};
```

\0

Note: **unsigned** is shorthand for **unsigned int**.

Subtypes

- on the arithmetic basic types **char**, **int**, **long**, etc. there exists a rich *subtype* hierarchy
- here $t_1 \leq t_2$, 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.

int ≤ double

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

int
double

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

extend the subtype relationship to more complex types

Example: Subtyping

Observe:

```
string extractInfo( struct { string info; } x) {
  return x.info;
}
```

u

- we would like `extractInfo` to be applicable to all argument records that contain a field `string info`
- use deduction rules to describe when $t_1 \leq t_2$ should hold
- the idea of subtyping on values is related to subtyping as implemented in object-oriented languages

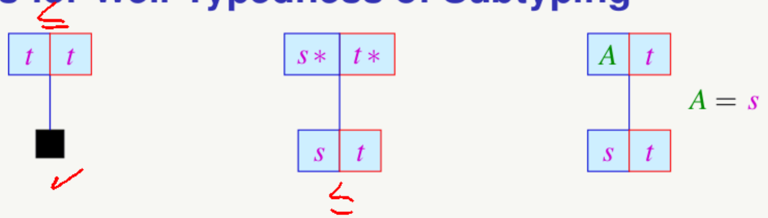
t = struct { string info; int value }

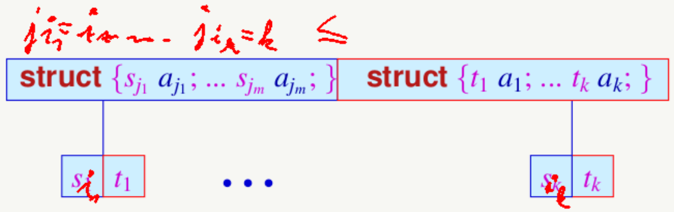
t ≤ u

int x;
double y;
y = x

int ≤ double

Rules for Well-Typedness of Subtyping



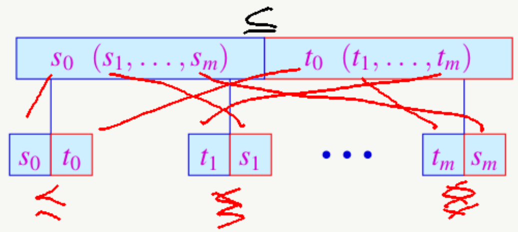


struct $\{int\ u, int\ v\}$ $\quad x;$

struct $\{int\ u\}$ $\quad y;$

$y = x;$

Rules and Examples for Subtyping



Examples:

int x (int);
float y (float);
y = x;

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

int ≤ float

int ≤ float

Attention:

- For functions:
- the return types are in normal subtype relationship
- for argument types, the subtype relation reverses

Co- and Contra Variance

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 $1 < i \leq n$

Example from function languages:

$$\textbf{int} \to (\textbf{float} \to \textbf{int}) \leq \textbf{int} \to (\textbf{int} \to \textbf{float})$$

$\iff$ $int \geq int \;\wedge$

$(float \to int) \geq (int \to float)$

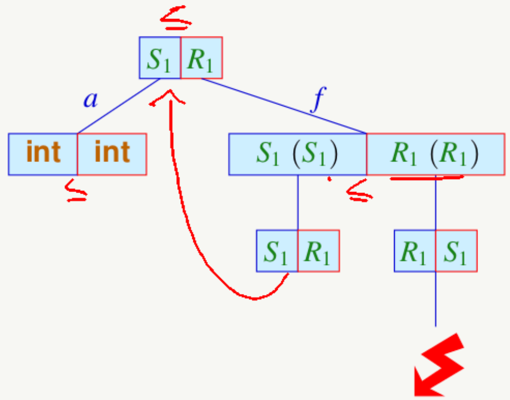
$\iff int \geq int \;\wedge\; int \leq float \;\wedge\; int \leq float$

These rules can be applied directly to test for sub-type relationship of recursive types

Subtypes: Application of Rules (I)

Check if $S_1 \leq R_1$:

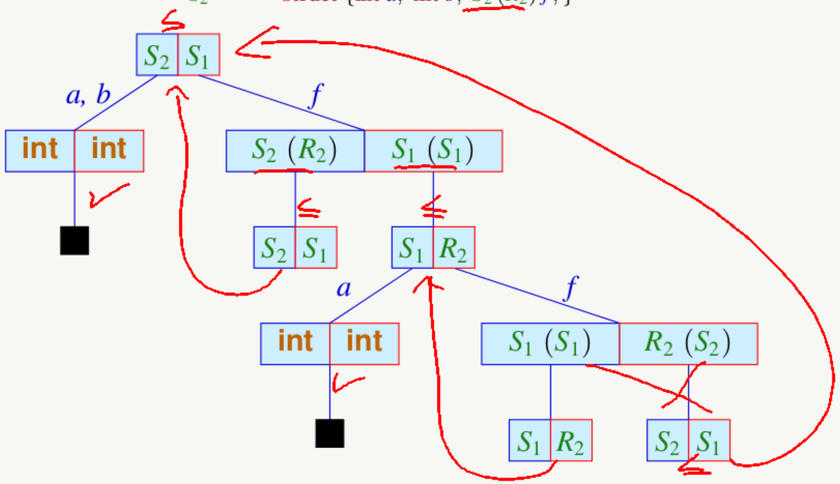
$$\begin{array}{lcl} R_1 & = & \textbf{struct}\ \{\textbf{int}\ a;\ R_1\ (R_1)\ f;\} \\ S_1 & = & \textbf{struct}\ \{\textbf{int}\ a;\ \textbf{int}\ b;\ S_1\ (S_1)\ f;\} \\ R_2 & = & \textbf{struct}\ \{\textbf{int}\ a;\ R_2\ (S_2)\ f;\} \\ S_2 & = & \textbf{struct}\ \{\textbf{int}\ a;\ \textbf{int}\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



Subtypes: Application of Rules (II)

Check if $S_2 \leq S_1$:

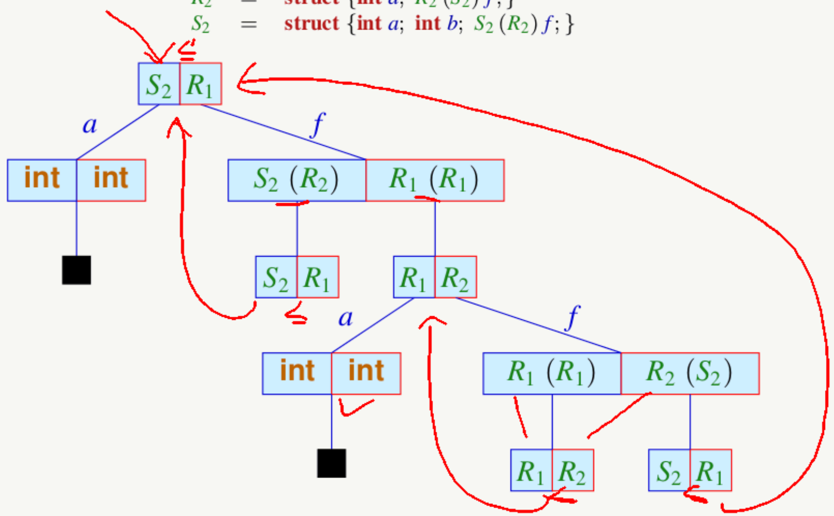
$$\begin{array}{lcl} R_1 & = & \textbf{struct}\ \{\textbf{int}\ a;\ R_1\ (R_1)\ f;\} \\ S_1 & = & \textbf{struct}\ \{\textbf{int}\ a;\ \textbf{int}\ b;\ S_1\ (S_1)\ f;\} \\ R_2 & = & \textbf{struct}\ \{\textbf{int}\ a;\ R_2\ (S_2)\ f;\} \\ S_2 & = & \textbf{struct}\ \{\textbf{int}\ a;\ \textbf{int}\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



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

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

30 / 30