École Polytechnique

CSC_52064 – Compilation

Jean-Christophe Filliâtre

static typing

if we write

```
"5" + 37
```

do we get

- a compile-time error? (OCaml, Rust, Go)
- a runtime error? (Python, Julia)
- the integer 42? (Visual Basic, PHP)
- the string "537"? (Java, Scala, Kotlin)
- a pointer? (C, C++)
- something else?

and what about

```
37 / "5"
```

?

if we now add two arbitrary expressions

```
e1 + e2
```

how can we decide whether this is legal and which operation to perform?

the answer is **typing**, a program analysis that binds **types** to each sub-expression, to rule out inconsistent programs

some languages are **dynamically typed**: types are bound to **values** and are used **at runtime**

examples: Lisp, PHP, Python, Julia

other languages are **statically typed**: types are bound to **expressions** and are used **at compile time**

examples: C, C++, Java, OCaml, Rust, Go

a language may use **both** static and dynamic typing

we will illustrate it with Java at the end of this lecture

today

1. static typing
   - 1.1 examples
   - 1.2 type checking WHILE, formally
   - 1.3 type safety
2. implementing type checking
3. subtyping
4. overloading

static typing

goals of typing

- type checking must be **decidable**
- type checking must reject programs whose evaluation would fail; this is **type safety**
- type checking must not reject too many non-absurd programs; the type system must be **expressive**

example

```
int f(long x) { return *x; }
```

```
file.c:1:23: error: invalid type argument of unary ‘*’
                    (have ‘long int’)
    2 | int f(long x) { return *x; }
      |                        ^~
```

but...

on the other hand, the compiler accepts an integer where a pointer is expected

```
int *f(long x) { return x; }
```

yet it emits a warning

```
file.c:1:24: warning: returning ‘long int’ from a function
             with return type ‘int *’ makes pointer from
             integer without a cast [-Wint-conversion]
    1 | int *f(long x) { return x; }
      |                        ^
```

one can "bypass" type checking with a **cast** (transtypage en français)

```
int f(long x) { return *((int*)x); }
```

the cast only impacts static typing and is a no-op in the generated code

```
f:
        movl    (%rdi), %eax
        ret
```

a C compiler introduce a lot of type conversions by itself,
notably between the various numerical types

examples:

```
int f(float x) { return x; }
float g(int x) { return x; }
```

here this means converting between integers and floating-point numbers

```
f:
        cvttss2sil      %xmm0, %eax
        ret
```

```
float f(int x, int y) { return x / y; }
```

divide then convert (implicit cast)

```
float f(int x, int y) { return ((float)x) / y; }
```

convert then divide

when the same name is used for several operations, we say it is **overloaded**

example: the same notation + can be used to add two integers, two floating-point numbers, a pointer and an integer, etc.

the **types of the operands**, set at compile time, determine which operation to perform

addition of two integers

```
int f(int x, int y) { return x + y; }
```

```
f: leal  (%rdi,%rsi), %eax
   ret
```

addition of a pointer and an integer (pointer arithmetic)

```
int *f(int *x, int y) { return x + y; }
```

```
f: movslq  %esi, %rsi
   leaq    (%rdi,%rsi,4), %rax
   ret
```

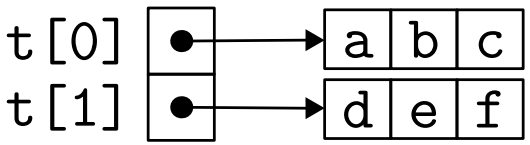
if the two operands must have the same type, but this is not the case, they are first **promoted** to a common type

the rules are complex (see for instance Kernighan & Ritchie)

another example

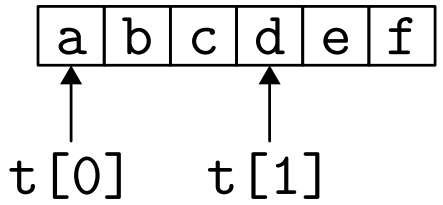
C makes a distinction between an array of pointers

```
int *t[2];
```



and an array of arrays

```
int t[2][3];
```



in both cases, we write `t[i][j]` and the type checking allows the compiler to emit different operations

and many other things

type checking also means checking

- the existence of types, of structure fields, etc.
- existence and uniqueness of functions, their arity
- existence of variables
- that a `break` is within a loop
- that `&e` mentions a left value
- etc.

limits of static typing

by nature, verification performed by static typing are limited to **decidable** properties

(reminder: decidable means that we can write a program that, for any input, terminates and outputs yes or no)

almost all "interesting" properties over source code are not decidable (Rice theorem)

the compiler cannot check

- the absence of division by zero
- that a computation terminates
- the absence of arithmetic overflow
- etc.

(and beyond that a program is doing what it is supposed to do!

yet it is possible

- to detect errors with certainty in some cases
- to emit warnings, at the risk of false positives

type checking WHILE, formally

let us consider the language WHILE from lecture 2

we are going to formally

1. give type checking rule for this language
2. show a type safety property

e	::=		**expression**
	|	n	constant (signed 32-bit integer)
	|	x	variable
	|	e *op* e	binary operator ($+$, $<\ldots$)
	|	`*`e	read from memory
s	::=		**statement**
	|	x`=`e`;`	assignment
	|	x`=malloc(4);`	allocate memory
	|	`*`e`=`e`;`	write to memory
	|	`if (`e`)` s `else` s	conditional
	|	`while (`e`)` s	loop
	|	`{`s `...` s`}`	block

we define a big-step operational semantics for expressions

$$M, E, e \twoheadrightarrow v$$

and a small-step operational semantics for statements

$$M, E, s \rightarrow M', E', s'$$

where E is a function from variables to values
and M is a function from addresses (ℓ) to integers (n)

v	$::=$		**value**
	$\mid$	n	integer value (signed 32-bit integer)
	$\mid$	ℓ	memory address

$$\frac{}{M, E, n \twoheadrightarrow n} \qquad \frac{x \text{ in } E}{M, E, x \twoheadrightarrow E(x)}$$

$$\frac{M, E, e_1 \twoheadrightarrow n_1 \quad M, E, e_2 \twoheadrightarrow n_2 \quad n \stackrel{\text{def}}{=} n_1 + n_2 \quad -2^{31} \leq n < 2^{31}}{M, E, e_1 + e_2 \twoheadrightarrow n} \qquad \text{etc.}$$

$$\frac{M, E, e \twoheadrightarrow \ell \quad \ell \text{ in } M}{M, E, *e \twoheadrightarrow M(\ell)}$$

$$\frac{M, E, e \twoheadrightarrow v}{M, E, x\texttt{=}e\texttt{;} \rightarrow M, E\{x \mapsto v\}, \texttt{\{ \}}} \qquad \frac{M, E, e_1 \twoheadrightarrow \ell \quad \ell \text{ in } M \quad M, E, e_2 \twoheadrightarrow n}{M, E, \texttt{*}e_1\texttt{=}e_2\texttt{;} \rightarrow M\{\ell \mapsto n\}, E, \texttt{\{ \}}}$$

$$\frac{\ell \text{ an address that is not in } M}{M, E, x\texttt{=malloc(4);} \twoheadrightarrow M\{\ell \mapsto n\}, E, \texttt{\{ \}}}$$

$$\frac{}{M, E, \texttt{\{ \{ \} } s \ldots \texttt{ \}} \rightarrow M, E, \texttt{\{} s \ldots \texttt{ \}}}$$

$$\frac{M, E, s_1 \rightarrow M_1, E_1, s_1'}{M, E, \texttt{\{} s_1 \ s_2 \ldots \texttt{ \}} \rightarrow M_1, E_1, \texttt{\{} s_1' \ s_2 \ldots \texttt{ \}}}$$

$$\frac{M, E, e \twoheadrightarrow n \neq 0}{M, E, \texttt{if (}e\texttt{)} \ s_1 \ \texttt{else} \ s_2 \rightarrow M, E, s_1} \qquad \frac{M, E, e \twoheadrightarrow 0}{M, E, \texttt{if (}e\texttt{)} \ s_1 \ \texttt{else} \ s_2 \rightarrow M, E, s_2}$$

$$\frac{M, E, e \twoheadrightarrow n \neq 0}{M, E, \texttt{while (}e\texttt{)} \ s \rightarrow M, E, \texttt{\{} s \ \texttt{while (}e\texttt{)} \ s \texttt{\}}}$$

$$\frac{M, E, e \twoheadrightarrow 0}{M, E, \texttt{while (}e\texttt{)} \ s \rightarrow M, E, \texttt{\{ \}}}$$

we introduce **types**, with the following abstract syntax

τ	$::=$		**type**
	$\mid$	`int`	type of integers
	$\mid$	`int*`	type of pointers

the type of a variable is given by a **typing environment** Γ
(a function from variables to types)

the **typing judgment** is written

$$\Gamma \vdash e : \tau$$

and reads "in typing environment Γ, expression e has type τ"

we use **inference rules** to define $\Gamma \vdash e : \tau$

$$\frac{}{\Gamma \vdash n : \texttt{int}}$$

$$\frac{x \in \Gamma}{\Gamma \vdash x : \Gamma(x)}$$

$$\frac{\Gamma \vdash e_1 : \texttt{int} \quad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 + e_2 : \texttt{int}} \qquad \text{etc.}$$

$$\frac{\Gamma \vdash e : \texttt{int*}}{\Gamma \vdash \texttt{*}e : \texttt{int}}$$

with $\Gamma = \{p \mapsto \texttt{int*};\ b \mapsto \texttt{int}\}$, we have

$$\dfrac{\dfrac{\dfrac{p \in \Gamma}{\Gamma \vdash p : \texttt{int*}}}{\Gamma \vdash \texttt{*}p : \texttt{int}} \quad \dfrac{b \in \Gamma}{\Gamma \vdash b : \texttt{int}}}{\Gamma \vdash \texttt{*}p + b : \texttt{int}}$$

this derivation is a proof that `*p+b` is well-typed

in the same environment, we cannot type expressions such as

$$*p + c$$

or

$$*42$$

or

$$1 + p$$

this is precisely what we want, for these expressions have no value in our semantics

to type statements, we introduce a new judgment

$$\Gamma \vdash s$$

that reads "in environment Γ, statement s is well-typed"

$$\frac{x \in \Gamma \quad \Gamma(x) = \tau \quad \Gamma \vdash e : \tau}{\Gamma \vdash x\texttt{=}e\texttt{;}}$$

$$\frac{x \in \Gamma \quad \Gamma(x) = \texttt{int*}}{\Gamma \vdash x\texttt{=malloc(4);}} \qquad \frac{\Gamma \vdash e_1 : \texttt{int*} \quad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash \texttt{*}e_1\texttt{=}e_2\texttt{;}}$$

$$\frac{\Gamma \vdash e : \tau \qquad \Gamma \vdash s_1 \qquad \Gamma \vdash s_2}{\Gamma \vdash \texttt{if (}e\texttt{)}\ s_1\ \texttt{else}\ s_2}$$

$$\frac{\Gamma \vdash e : \tau \qquad \Gamma \vdash s}{\Gamma \vdash \texttt{while (}e\texttt{)}\ s}$$

$$\frac{}{\Gamma \vdash \texttt{\{ \}}} \qquad \frac{\Gamma \vdash s_1 \quad \Gamma \vdash \texttt{\{}\ s_2 \ldots \texttt{\}}}{\Gamma \vdash \texttt{\{}\ s_1\ s_2 \ldots \texttt{\}}}$$

well-typed programs do not go wrong

(Milner, 1978)

let us show that our type system is safe wrt our operational semantics

Theorem (type safety)

If $\Gamma \vdash s$, then the reduction s is either infinite or reaches $\{\ \}$.

or, equivalently,

Theorem

If $\Gamma \vdash s$ and $M, E, s \to^\star M', E', s'$ and s' is irreducible, then s' is $\{\ \}$.

this means evaluation won't be **stuck** or any expression such as

$$*42$$

or on a statement

$$\texttt{if (}e\texttt{) } s_1 \texttt{ else } s_2$$

where e does not evaluate to a value

let us show first that well-typed expressions do evaluate successfully

$$\text{if } \Gamma \vdash e : \tau, \text{ then } M, E, e \twoheadrightarrow v$$

stated as such, this is not correct: we need a relationship between Γ on one side and M and E on the other side

counterexamples:

$$\begin{array}{rcl} \Gamma & = & \{p \mapsto \texttt{int*}\} \\ e & = & *p \\ E & = & \{p \mapsto 89\} \\ M & = & \ldots \end{array} \qquad \begin{array}{rcl} \Gamma & = & \{p \mapsto \texttt{int*}\} \\ e & = & *p \\ E & = & \{p \mapsto \ell\} \\ M & = & \emptyset \end{array}$$

Definition (well-typed environment)

An execution environment M, E is well-typed in a typing environment Γ, written $\Gamma \vdash M, E$, if

$$\forall x,\ \textit{if}\ \Gamma(x) = \tau\ \textit{then}\ \begin{cases} x \in E\ \textit{and}\ \Gamma \vdash E(x) : \tau, \\ \textit{if}\ \tau = \texttt{int*}\ \textit{then}\ E(x) \in M. \end{cases}$$

Lemma (evaluation of a well-typed expression)

If $\Gamma \vdash e : \tau$ *and* $\Gamma \vdash M, E$, *then* $M, E, e \twoheadrightarrow v$ *and* $\Gamma \vdash v : \tau$.
Beside, if $v = \ell$ *then* $\ell \in M$.

proof: by induction on the derivation $\Gamma \vdash e : \tau$.

- $e = n$ immediate with $v = n$
- $e = x$ immediate with $v = E(x)$;
  and if $v = \ell$ then $\ell \in M$ since M, E is well-typed
- $e = e_1 + e_2$ by IH on e_1 and e_2 we have $M, E, e_i \twoheadrightarrow v_i$ and $\Gamma \vdash v_i : \texttt{int}$,
  so v_1 and v_2 are integers and we conclude with $v = v_1 + v_2$
- e=*e_1 by IH on e_1 we have $M, E, e_1 \twoheadrightarrow \ell$ and since M, E is
  well-typed we have $\ell \in M$, so we conclude with $v = M(\ell)$.

□

the type safety proof is based on two lemmas

Lemma (progress)

If $\Gamma \vdash s$ *and* $\Gamma \vdash M, E$, *then either* s *is* $\{\ \}$, *or* $M, E, s \to M', E', s'$.

Lemma (preservation)

If $\Gamma \vdash s$, *if* $\Gamma \vdash M, E$ *and if* $M, E, s \to M', E', s'$ *then* $\Gamma \vdash s'$ *and* $\Gamma \vdash M', E'$.

Lemma (progress)

If $\Gamma \vdash s$ *and* $\Gamma \vdash M, E$, *then either* s *is* $\{\ \}$, *or* $M, E, s \to M', E', s'$.

proof: by induction on the derivation $\Gamma \vdash s$

$s = \{\ \}$ immediate

$s = \{\ s_1\ s_2 \dots\ \}$ if $s_1 = \{\ \}$, we have $M, E, \{\ s_1\ s_2 \dots\ \} \to M, E, \{\ s_2 \dots\ \}$
otherwise, we use IH on s_1, so $M, E, s_1 \to M', E', s_1'$ and thus $M, E, \{\ s_1\ s_2 \dots\ \} \to M', E', \{\ s_1'\ s_2 \dots\ \}$

$s = *e_1 = e_2\,;$ since e_1 and e_2 are well-typed, they evaluate to ℓ and n respectively, with $\ell \in M$, and thus $M, E, *e_1 = e_2\,; \to M\{\ell \mapsto n\}, E, \{\ \}$

other cases left as exercise □

then we show

Lemma (preservation)

If $\Gamma \vdash s$, *if* $\Gamma \vdash M, E$ *and if* $M, E, s \to M', E', s'$ *then* $\Gamma \vdash s'$ *and* $\Gamma \vdash M', E'$.

proof: by induction on the derivation $\Gamma \vdash s$

$s = \{\ s_1\ s_2 \dots\ \}$ we have $\Gamma \vdash s_1$ and $\Gamma \vdash \{\ s_2 \dots\ \}$

- if $s_1 = \{\ \}$, then $M, E, \{\ s_1\ s_2 \dots\ \} \to M, E, \{\ s_2 \dots\ \}$
- otherwise, $M, E, s_1 \to M', E', s_1'$ and by IH $\Gamma \vdash s_1'$ and $\Gamma \vdash M', E'$ so $\Gamma \vdash \{\ s_1'\ s_2 \dots\ \}$

$s = *e_1 = e_2\ ;$ we have $M, E, e_1 \twoheadrightarrow \ell$ and $M, E, e_2 \twoheadrightarrow n$ (slide 40) and $s' = \{\ \}$ (so $\Gamma \vdash s'$) and $M' = M\{\ell \mapsto n\}$
thus $\Gamma \vdash s'$ and $\Gamma \vdash M', E'$

other cases left as exercise □

now we can deduce type safety easily

Theorem (type safety)

If $\Gamma \vdash s$ *and* $\Gamma \vdash M, E$ *and* $M, E, s \to^\star M', E', s'$ *and* s' *is irreducible, then* s' *is* $\{\ \}$.

proof: we have $M, E, s \to M_1, E_1, s_1 \to \cdots \to M', E', s'$ and by repeated applications of the preservation lemma, we have $\Gamma \vdash s'$
by the progress lemma, s' is reducible or is $\{\ \}$
so this is $\{\ \}$ □

the lecture notes contain a similar proof for Mini-ML,
with types as follows

$$
\begin{array}{rcll}
\tau & ::= & \texttt{int} \mid \texttt{bool} \mid \ldots & \textit{base types} \\
& \mid & \tau \to \tau & \textit{function type} \\
& \mid & \tau \times \tau & \textit{type of a pair}
\end{array}
$$

as we just did, the proof is based on progress and preservation properties

see chapter 5

languages such as Java or OCaml enjoy such a type safety property

which means that the evaluation of an expression of type τ

- either does not terminate
- or raises an exception
- or terminates on a value **with type τ**

in OCaml, the absence of `null` makes it a rather strong property

implementing type checking

there is a difference between **the typing rules**, which define the relation

$$\Gamma \vdash e : \tau$$

and the **type checking algorithm**, which checks that a given expression e is well-typed in some environment Γ

for instance

- the type τ is not necessarily given (type inference)
- several rules may apply for a single construct
- an expression may have several types

the case of WHILE is simple, as a single rule applies for each expression

we say that typing is **syntax-directed**

the type checking is then implemented with a linear time traversal of the program

- we do not simply say

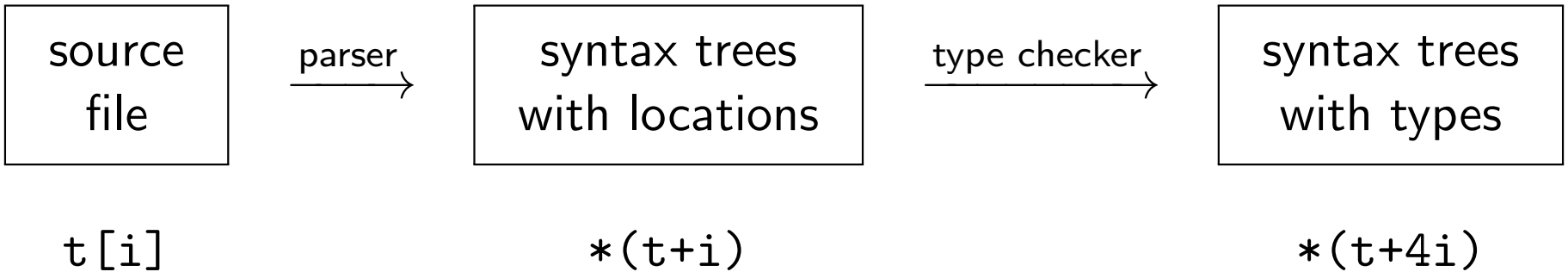
```
type error
```

  but we **explain** the type error precisely

- we **keep** types for the further phases of the compiler

to do this, we **decorate** abstract syntax trees

- **input** of type checking contains positions in source code
- **output** of type checking contains types



decorated AST

in OCaml

```
type loc = ...
```

```
type expr =
| Evar   of string
| Econst of int
| Efield of expr * string
...
```

in Java

```
class Loc { ... }
```

```
abstract class Expr {
}
class Evar   extends Expr {...}
class Econst extends Expr {...}
class Efield extends Expr {...}
...
```

decorated AST

in OCaml

```
type loc = ...
```

```
type expr = {
  desc: desc;
  loc : loc;
}
and desc =
| Evar   of string
| Econst of int
| Efield of expr * string
...
```

in Java

```
class Loc { ... }
```

```
abstract class Expr {
  Loc loc;


}
class Evar   extends Expr {...}
class Econst extends Expr {...}
class Efield extends Expr {...}
...
```

we signal a type error with an exception

the exception contains

- a message explaining the error
- a position in the source code

we catch this exception in the main function

we display the position and the message

```
test.c:8:14: error: too few arguments to function 'f'
```

we set up an abstract syntax for types

```
type typ = ...
```

```
class Typ { ... }
```

and a **new** abstract syntax for programs

```
type texpr = {
  tdesc: tdesc;
  typ  : typ
}
and tdesc =
| Tvar   of string
| Tconst of int
| Tfield of texpr * string
...
```

```
abstract class Texpr {
  Typ typ;


}
class Tvar   extends Texpr {...}
class Tconst extends Texpr {...}
class Tfield extends Texpr {...}
...
```

during type checking, the compiler maintains data structures that contain the symbols in scope

we have a dictionary for variables, another for types, etc.

these are called **name spaces**

the following C program is accepted (even if questionable)

```
struct f { int f; };
int f(struct f f) { f.f = 1; }
```

but the following is rejected

```
int f(int f) { return f(f); }
```

at any moment, we know the variables that are in scope

for each one, we know

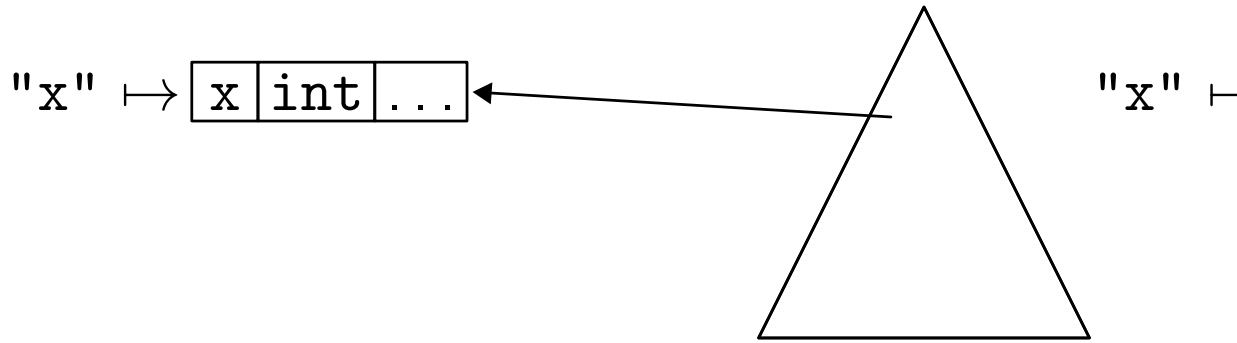
- the location of its declaration
- its type
- possibly other things (size in memory, etc.)

terminology: **symbol tables** refer to such data associated to symbols inside compilers

a variable may temporarily hide another one; this is called **shadowing**

```
int x = 1;
{ int *x; ... }
x = 2; // back with int x
```

```
int x;
x = 1;
{ int *x; ... }
x = 2;
```



during type checking, a dictionary maps names to symbolswhen we are done, we throw the dictionary away, and we are left with the typed tree containing the symbols

subtyping

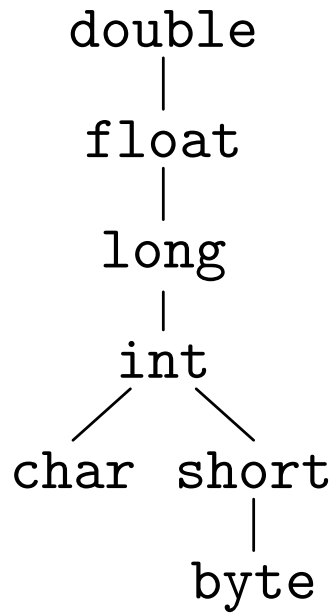
we say that a type τ_1 is a **subtype** of a type τ_2, which we write

$$\tau_1 \leq \tau_2$$

if any value with type τ_1 can be considered as a value with type τ_2

in many languages, there is subtyping between numerical types

in Java, it is as shown on the right



thus we can write

```
int n = 'a';
```

but not

```
byte b = 144;
```

in an object-oriented language, inheritance induces **subtyping**:
if a class `B` inherits from a class `A`, we have

$$\texttt{B} \leq \texttt{A}$$

i.e. any value of type `B` can be seen as a value of type `A`

the two classes

```
class Vehicle             { ... void move() { ... } ... }
class Car extends Vehicle { ... void move() { ... } ... }
```

induce the subtyping relation

$$\texttt{Car} \leq \texttt{Vehicle}$$

and thus we can write

```
Vehicle v = new Car();
v.move();
```

the construct `new C(...)` builds an object of class `C`, and the class of this object cannot be changed in the future; this is the **dynamic type** of the object

however, the **static type** of an expression, as computed by the compiler, may differ from the dynamic type, because of subtyping

when we write

```
Vehicle v = new Car();
v.move();
```

variable `v` has type `Vehicle`, but the method `move` that is called is that of class `Car` (we'll explain how in another lecture)

in many cases, the compiler **cannot** determine the dynamic type

example:

```
void moveAll(LinkedList<Vehicule> l) {
  for (Vehicule v: l)
    v.move();
}
```

sometimes we need to force the compiler's hand, which means we claim that a value has some type

we call this **type casting** (or simply `cast`)

Java's notation, inherited from C, is

$$(\tau)e$$

the static type of this expression τ

using a cast, we can write

```
int  n = ...;
byte b = (byte)n;
```

in this case, there is no dynamic verification
(if the integer is too large, it is truncated)

let us consider

$$(C)e$$

where

- D is the dynamic type of (the object designated by) e
- E is the static type of expression e

there are three cases

- C is a super class of E: this is an **upcast** and the code for $(C)e$ is that of e (but the cast has some influence anyway, since $(C)e$ has type C)
- C is a subclass of E: this is a **downcast** and the code contains **dynamic test** to check that D is indeed a subclass of C
- C is neither a subclass nor a super of E: the compiler rejects the program with a type error

example (upcast)

```
class A {
  int x = 1;
}

class B extends A {
  int x = 2;
}
```

```
  B b = new B();
  System.out.println(b.x);       // 2
  System.out.println(((A)b).x); // 1
```

example (downcast)

```
void m(Vehicle v, Vehicle w) {
  ((Car)v).await(w);
}
```

nothing guarantees that the object passed to `m` will be a car; in particular, it could have no method `await`!

the dynamic test is required

Java raises `ClassCastException` is the test fails

to allow defensive programming, there exists a Boolean construct

$$e \texttt{ instanceof } C$$

that checks whether the class of e is indeed a subclass of C

it is idiomatic to do

```
if (e instanceof C) {
  C c = (C)e;
  ...
}
```

in this case, the compiler makes an optimization to perform a single test

overloading

(**surcharge** en français)

overloading is the ability to reuse the same name of several operations

overloading is handled **at compile time**, using the number and the (static) types of arguments

caveat: not to be confused with *overriding* (see lecture 6)

in Java, operation + is overloaded

```
int    n = 40 + 2;
String s = "foo" + "bar";
String t = "foo" + 42;
```

these are three distinct operations

```
int    +(int   , int   )
String +(String, String)
String +(String, int   )
```

when we write

```
int n = 'a' + 42;
```

this is subtyping that allows us to consider `'a'` with type `char` as a value of type `int`, and thus the operation is `+(int, int)`

but when we write

```
String t = "foo" + 42;
```

this is **not** subtyping (`int` $\not\leq$ `String`)

in particular, we cannot write

```
String t = 42;
```

in Java, one cannot overload operators such as +
but one can overload methods/constructors

```
int f(int n, int m) { ... }
int f(int n)        { ... }
int f(String s)     { ... }
```

this is exactly as if we had written

```
int f_int_int(int n, int m) { ... }
int f_int    (int n)        { ... }
int f_String (String s)     { ... }
```

the compiler uses the static types of `f`'s arguments to determine which method to call

yet overloading resolution can be tricky

```
class A {...}
class B extends A {
  void m(A a) {...}
  void m(B b) {...}
}
```

with

```
{ ... B b = new B(); b.m(b); ... }
```

both methods apply

this is method `m(B b)` that is called, because it is considered **more precise**

some cases are ambiguous

```
class A {...}
class B extends A {
  void m(A a, B b) {...}
  void m(B b, A a) {...}
}
{ ... B b = new B(); b.m(b, b); ... }
```

and reported as such

```
test.java:13: reference to m is ambiguous,
  both method m(A,B) in B and method m(B,A) in B match
```

to each method defined in class C

$$\tau\ \mathtt{m}(\tau_1\ x_1, ..., \tau_n\ x_n)$$

we set the profile $(\mathtt{C}, \tau_1, \ldots, \tau_n)$

then we **order** profiles: $(\tau_0, \tau_1, \ldots, \tau_n) \sqsubseteq (\tau'_0, \tau'_1, \ldots, \tau'_n)$ if and only if τ_i is a subtype of τ'_i for all i

for a call

$$e.m(e_1, \ldots, e_n)$$

where e has static type τ_0 and e_i has static type τ_i, we consider the set of all **minimal** elements in the set of all compatible profiles

- no element $\Rightarrow$ no method applies
- several elements $\Rightarrow$ ambiguity
- a single element $\Rightarrow$ this is the method to call

as far as the execution model is concerned, Python is relatively close to Java, but there is almost **no static typing** in Python

all the verification is performed **dynamically**

including the existence of a function, the check of its arity, etc.

- labs 4–9 = the project = a tiny Java to x86-64 compiler
  - PDF description on Moodle

- lab 4 = static typing of Mini Java
  - the parser is provided
    - but no need to run CUP or Menhir
  - both parsed **and typed** AST are provided
    - **read and understand before you start**
  - tests are provided
    - **test as you go**

- next lecture
  - evaluation strategy
  - parameter passing