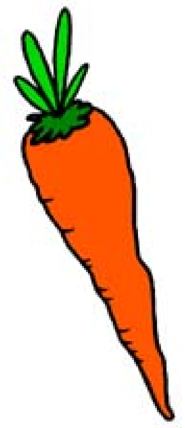
Languages and Compilation

Based on the Jean-Christophe Filliâtre's Courses
given at École Polytechnique & École Normale Supérieure

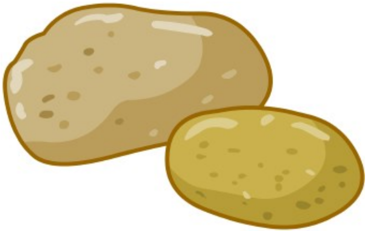
Lecture 6 - Typing

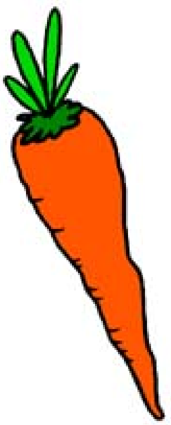
Léon Gondelman

AALBORG UNIVERISITY | COPENHAGEN | 2025

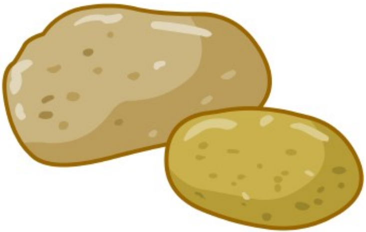


+





/



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

when?

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

consider the following C and Python code snippets:

```c
int id(int num) {
  return num; }

void main(){
  printf("%d", id(42,42));}
```

```python
def id(num):
  return num

print(id(42,42))
```

the C code fails at the compile-time (compilation error)
error: too many arguments to function 'id'

the Python code compiles to the VM and fails at runtime (runtime error)
TypeError: id() takes 1 positional argument but 2 were given

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

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

- **lecture:**
  - static typing, illustrated on WHILE with record types
  - type safety
  - implementing type checking algorithm
  - subtyping and overloading

- **lab session:**
  - static type checking a fragment of C
  - covers type-checking struct pointers and function declarations

static typing

slogan (Milner, 1978)

well-typed programs do not go wrong

- 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**

1. any sub-expression is annotated with a type

```
int f(int x) { int y = ((x:int)+(1:int):int); ... }
```

type checking is easy but this is unmanageable for the programmer

2. only annotate variable declarations (C, C++, Java, etc.)

```
int f(int x) { int y = x+1; return y; }
```

3. only annotate function parameters (C++ 11, Java 10)

```
int f(int x) { var y = x+1; return y; }
```

4. no annotation at all ⇒ **type inference** (OCaml, Haskell, etc.)

```
fun x -> x+1
```

let us consider the language WHILE from lecture 2

to make it more interesting, let us add **records**
(and any variable is a record)

note: for simplicity, here we consider **anonymous** records; in languages like C, records are named and record fields are declared with their types in the source program

(see the Lab session on type checking a fragment of C).

e	$::=$		**expression**
	$\mid$	c	integer or Boolean constant
	$\mid$	x	variable
	$\mid$	$e.f$	field access
	$\mid$	$e\ op\ e$	binary operator $(+, <, \dots)$
s	$::=$		**statement**
	$\mid$	$e.f \leftarrow e$	assignment
	$\mid$	$\texttt{if}\ e\ \texttt{then}\ s\ \texttt{else}\ s$	conditional
	$\mid$	$\texttt{while}\ e\ \texttt{do}\ s$	loop
	$\mid$	$s;\ s$	sequence
	$\mid$	$\texttt{skip}$	do nothing

$$
\begin{array}{l}
x.a \leftarrow 0; \\
x.b \leftarrow 1; \\
\texttt{while } x.b < 100 \texttt{ do} \\
\quad x.b \leftarrow x.a + x.b; \\
\quad x.a \leftarrow x.b - x.a
\end{array}
$$

the notion of value from lecture 2 is updated

$$
\begin{array}{rcll}
v & ::= & & \textbf{value} \\
 & | & n & \text{integer value} \\
 & | & b & \text{Boolean value} \\
 & | & x & \text{address (here the name of the variable)}
\end{array}
$$

we also update the environment E,
which now maps pairs (x, f) to values $E(x, f)$

we define a big-step operational semantics for expressions

$$E, e \twoheadrightarrow v$$

and a small-step operational semantics for statements

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

$$\frac{}{E, n \twoheadrightarrow n} \qquad \frac{}{E, b \twoheadrightarrow b}$$

$$\frac{}{E, x \twoheadrightarrow x}$$

$$\frac{E, e \twoheadrightarrow x \quad (x, f) \in \mathrm{dom}(E)}{E, e.f \twoheadrightarrow E(x, f)}$$

$$\frac{E, e_1 \twoheadrightarrow n_1 \quad E, e_2 \twoheadrightarrow n_2 \quad n = n_1 + n_2}{E, e_1 + e_2 \twoheadrightarrow n} \qquad \text{etc.}$$

$$\frac{E, e_1 \twoheadrightarrow x \qquad E, e_2 \twoheadrightarrow v \quad (x, f) \in \mathrm{dom}(E)}{E, e_1.f \leftarrow e_2 \rightarrow E\{(x, f) \mapsto v\}, \texttt{skip}}$$

$$\frac{}{E, \texttt{skip}; s \rightarrow E, s} \qquad \frac{E, s_1 \rightarrow E_1, s_1'}{E, s_1; s_2 \rightarrow E_1, s_1'; s_2}$$

$$\frac{E, e \twoheadrightarrow \texttt{true}}{E, \texttt{if } e \texttt{ then } s_1 \texttt{ else } s_2 \rightarrow E, s_1} \qquad \frac{E, e \twoheadrightarrow \texttt{false}}{E, \texttt{if } e \texttt{ then } s_1 \texttt{ else } s_2 \rightarrow E, s_2}$$

$$\frac{E, e \twoheadrightarrow \texttt{true}}{E, \texttt{while } e \texttt{ do } s \rightarrow E, s; \texttt{while } e \texttt{ do } s}$$

$$\frac{E, e \twoheadrightarrow \texttt{false}}{E, \texttt{while } e \texttt{ do } s \rightarrow E, \texttt{skip}}$$

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

τ	$::=$		**type**
	$\mid$	`int`	type of integer values
	$\mid$	`bool`	type of Boolean values
	$\mid$	$\{f : \tau; \dots; f : \tau\}$	record type

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 $\Gamma \vdash e : \tau$

$$\frac{}{\Gamma \vdash n : \texttt{int}} \qquad \frac{}{\Gamma \vdash b : \texttt{bool}}$$

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

$$\frac{\Gamma \vdash e : \{\dots; f : \tau; \dots\}}{\Gamma \vdash e.f : \tau}$$

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

with $\Gamma = \{x \mapsto \{a : \texttt{int}; b : \texttt{int}\}\}$, we have

$$\dfrac{\dfrac{\overline{\Gamma \vdash x : \{a : \texttt{int}; b : \texttt{int}\}}}{\Gamma \vdash x.a : \texttt{int}} \qquad \overline{\Gamma \vdash 1 : \texttt{int}}}{\Gamma \vdash x.a + 1 : \texttt{int}}$$

this derivation is a proof that `x.a+1` is well-typed in the same environment, we cannot type expressions such as

$$x.c$$

or

$$42.a$$

or

$$1 + \mathtt{true}$$

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"

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

$$\frac{\Gamma \vdash e_1 : \{\ldots; f : \tau : \ldots\} \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1.f \leftarrow e_2}$$

$$\frac{\Gamma \vdash e : \texttt{bool} \qquad \Gamma \vdash s_1 \qquad \Gamma \vdash s_2}{\Gamma \vdash \texttt{if } e \texttt{ then } s_1 \texttt{ else } s_2}$$

$$\frac{\Gamma \vdash e : \texttt{bool} \qquad \Gamma \vdash s}{\Gamma \vdash \texttt{while } e \texttt{ do } s}$$

well-typed programs do not go wrong

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 `skip`.

or, equivalently,

Theorem

If $\Gamma \vdash s$ and $E, s \rightarrow^{\star} E', s'$ and s' is irreducible, then s' is `skip`.

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

$$42.a$$

or on a statement

$$\texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2$$

where e does not evaluate to either `true` or `false` let us show first that well-typed expressions do evaluate successfully

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

stated as such, this is not correct: we need a relationship between Γ and E

counterexample:

$$\begin{array}{rcl} \Gamma & = & \{x \mapsto \{a : \mathtt{int}\}\} \\ e & = & x.a \\ E & = & \emptyset \end{array}$$

Definition (well-typed environment)

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

$$\forall x,\ \textit{if}\ \Gamma(x) = \{\ldots f : \tau \ldots\}\ \textit{then}\ (x, f) \in \textit{dom}(E)\ \textit{and}\ \Gamma \vdash E(x, f) : \tau.$$

Lemma (evaluation of a well-typed expression)

If $\Gamma \vdash e : \tau$ and $\Gamma \vdash E$, then $E, e \twoheadrightarrow v$ and $\Gamma \vdash v : \tau$.

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

$e = c$ immediate with $v = c$

$e = x$ immediate with $v = x$

$e = e_1.f$ by IH $E, e_1 \twoheadrightarrow v_1$ and $\Gamma \vdash v_1 : \tau_1$ with $\tau_1 = \{\dots f : \tau \dots\}$.
so v_1 is an address x and $v = E(x, f)$
since E is well-typed, we have $\Gamma \vdash v : \tau$

$e = e_1 + e_2$ by IH on e_1 and e_2 we have $E, e_i \twoheadrightarrow v_i$ and $\Gamma \vdash v_i : \mathtt{int}$,
so v_1 and v_2 are integers and we conclude with $v = v_1 + v_2$

$\square$

the type safety proof is based on two lemmas

Lemma (progress)

If $\Gamma \vdash s$ *and* $\Gamma \vdash E$*, then either* s *is* `skip`*, or* $E, s \rightarrow E', s'$.

Lemma (preservation)

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

Lemma (progress)

If $\Gamma \vdash s$ and $\Gamma \vdash E$, then either s is `skip`*, or $E, s \rightarrow E', s'$.*

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

$s = \texttt{skip}$ immediate

$s = s_1; s_2$ if $s_1 = \texttt{skip}$, we have $E, s_1; s_2 \rightarrow E, s_2$
otherwise, we use IH on s_1, so $E, s_1 \rightarrow E', s_1'$ and thus
$E, s_1; s_2 \rightarrow E', s_1'; s_2$

$s = e_1.f \leftarrow e_2$ since e_1 and e_2 are well-typed, they evaluate to x and v respectively
since $\Gamma \vdash x : \{\dots f : \tau \dots\}$ we have $(x, f) \in \mathrm{dom}(E)$ and thus $E, s \rightarrow E', \texttt{skip}$ with $E' = E\{(x, f) \mapsto v\}$

other cases left as exercise □

then we show

Lemma (preservation)

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

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

$s = s_1; s_2$ we have $\Gamma \vdash s_1$ and $\Gamma \vdash s_2$

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

$s = e_1.f \leftarrow e_2$ we have $E, e_1 \twoheadrightarrow x$ and $E, e_2 \twoheadrightarrow v$ and $s' = \mathtt{skip}$ (so $\Gamma \vdash s'$) and $E' = E\{(x, f) \mapsto v\}$ but $\Gamma \vdash e_1 : \{\dots f : \tau \dots\}$ and $\Gamma \vdash e_2 : \tau$ so $\Gamma \vdash v : \tau$ (see slide 33) and thus $\Gamma \vdash E'$

other cases left as exercise □

now we can deduce type safety easily

Theorem (type safety)

If $\Gamma \vdash s$ and $E, s \rightarrow^\star E', s'$ and s' is irreducible, then s' is `skip`.

proof: we have $E, s \rightarrow E_1, s_1 \rightarrow \cdots \rightarrow E', s'$ and by repeated applications of the preservation lemma, we have $\Gamma \vdash s'$
by the progress lemma, s' is reducible or is `skip`
so this is `skip` $\square$

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

source file $\xrightarrow{\text{parser}}$ syntax trees with locations $\xrightarrow{\text{type checker}}$ syntax trees with types 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 {...}
...
```

in OCaml

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

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

in Java

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

```java
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 {...}
...
```

the type checker turns a parsed syntax tree into **another**, typed syntax tree

$$\boxed{\text{parsed trees}} \xrightarrow{\text{type checker}} \boxed{\text{typed trees}}$$

yet this is efficient, since

- it is typically a linear traversal
- former AST are collected by the GC

subtyping

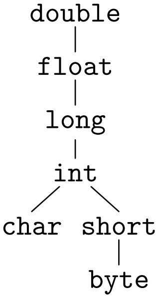
we say that a type τ_1 is a **subtype** of a type τ_2, which we write

$$\tau_1 \leq \tau_2$$

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

$$\mathtt{B} \leq \mathtt{A}$$

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 is 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

```
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
  b.x = 4;
  ((A)b).x = 3;
  System.out.println(b.x);       // 4
  System.out.println(((A)b).x); // 3
```

```
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` if the test fails

```
class A { int x = 1; }
class B extends A { int x = 2; }

class Example{
    static A a = new A();
    static B b = new B();
    static int m (A a){
      return ((B)a).x; }

public static void main(String args[]){
//   System.out.println(m(a)); // runtime error
     System.out.println(m(b)); // 2
    }
 }
```

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

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 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)`

for instance, `System.out.println('m' - 'n');` will output `-1`

but when we write

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

this is **not** subtyping (`int` $\not\leq$ `String`) but is due to two built-in '$+$'

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 $\mathtt{C}$

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

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