École Polytechnique

INF564 – Compilation

Jean-Christophe Filliâtre

typage

si j'écris l'expression

```
    "5" + 37
```

dois-je obtenir

- une erreur à la compilation ? (OCaml, Rust)
- une erreur à l'exécution ? (Python)
- l'entier 42 ? (Visual Basic, PHP)
- la chaîne "537" ? (Java)
- un pointeur ? (C, C++)
- autre chose encore ?

et qu'en est-il de

```
  37 / "5"
```

?

et si on additionne deux expressions arbitraires

```
e1 + e2
```

comment déterminer si cela est légal et ce que l'on doit faire le cas échéant ?

la réponse est le **typage**, une analyse qui associe un **type** à chaque sous-expression, dans le but de rejeter les programmes incohérents

certains langages sont **typés dynamiquement** : les types sont associés aux **valeurs** et utilisés pendant l'**exécution** du programme

exemples : Lisp, PHP, Python, Julia

d'autres sont **typés statiquement** : les types sont associés aux **expressions** et utilisés pendant la **compilation** du programme

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

c'est ce second cas que l'on considère dans ce cours

slogan (Milner, 1978)

well-typed programs do not go wrong

- le typage doit être **décidable**
- le typage doit rejeter les programmes absurdes dont l'évaluation échouerait ; c'est la **sûreté du typage**
- le typage ne doit pas rejeter trop de programmes non-absurdes, *i.e.* le système de types doit être **expressif**

1. toutes les sous-expressions sont annotées par un type

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

   facile à vérifier mais trop fastidieux pour le programmeur

2. annoter seulement les déclarations de variables (C, C++, Java, etc.)

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

3. annoter seulement les paramètres de fonctions (C++ 11, Java 10)

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

4. ne rien annoter ⇒ **inférence** complète (OCaml, Haskell, etc.)

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

un algorithme de typage doit avoir les propriétés de

- **correction** : si l'algorithme répond "oui" alors le programme est effectivement bien typé
- **complétude** : si le programme est bien typé, alors l'algorithme doit répondre "oui"

et éventuellement de

- **principalité** : le type calculé pour une expression est le plus général possible

reprenons le langage WHILE du cours 2

pour le rendre un peu plus intéressant, on y ajoute des **enregistrements** (et on suppose maintenant que les variables dénotent toutes des enregistrements)

e	$::=$		**expression**
	$\mid$	c	constante entière ou booléenne
	$\mid$	x	variable
	$\mid$	$e.f$	accès à un champ
	$\mid$	$e\ op\ e$	opérateur binaire $(+, <, \dots)$
s	$::=$		**instruction**
	$\mid$	$e.f \leftarrow e$	affectation
	$\mid$	if e then s else s	conditionnelle
	$\mid$	while e do s	boucle
	$\mid$	$s;\ s$	séquence
	$\mid$	skip	ne rien faire

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

la notion de valeur est légèrement modifiée

v	$::=$		**valeur**
	$\mid$	n	valeur entière
	$\mid$	b	valeur booléenne
	$\mid$	x	adresse (ici le nom de la variable)

ainsi que la notion d'environnement E,
qui associe une valeur $E(x, f)$ à certains couples (x, f)

on définit une sémantique à grands pas pour les expressions

$$E, e \twoheadrightarrow v$$

et une sémantique à petits pas pour les instructions

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

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

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

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

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

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

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

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

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

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

on se donne des **types**, dont la syntaxe abstraite est

τ	$::=$		**type**
	$\mid$	`int`	entier
	$\mid$	`bool`	booléen
	$\mid$	$\{f : \tau; \dots; f : \tau\}$	enregistrement

le type d'une variable est donné par un **environnement** Γ
(une fonction des variables vers les types)

le **jugement** de typage que l'on va définir se note

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

et se lit « dans l'environnement Γ, l'expression e a le type τ »

on utilise des règles d'inférence pour définir $\Gamma \vdash e : \tau$

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

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

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

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

en posant $\Gamma = \{x \mapsto \{a : \texttt{int}; b : \texttt{int}\}\}$, on a

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

en revanche, on ne peut pas typer des expressions comme

$$x.c$$

ou bien

$$42.a$$

ou encore

$$1 + \texttt{true}$$

c'est précisément l'objectif, car ces expressions n'ont pas de valeur dans notre sémantique

pour typer les instructions, on introduit un autre jugement

$$\Gamma \vdash s$$

qui se lit « dans l'environnement Γ, l'instruction s est bien typée »

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

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

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

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

montrons l'adéquation du typage par rapport à la sémantique à réductions

Théorème (sûreté du typage)

Si $\Gamma \vdash s$*, alors la réduction de* s *est infinie ou termine sur* `skip`.

ou, de manière équivalente,

Théorème

Si $\Gamma \vdash s$ *et* $E, s \rightarrow^\star E', s'$ *et* s' *irréductible, alors* s' *est* `skip`.

cela veut dire que l'évaluation ne se retrouvera pas **bloquée** sur une expression telle que

$$42.a$$

ou encore sur une instruction

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

où e ne s'évalue ni en `true` ni en `false`

montrons en premier lieu qu'une expression bien typée s'évalue correctement, c'est-à-dire

$$\text{si } \Gamma \vdash e : \tau, \text{ alors } E, e \twoheadrightarrow v$$

dit comme cela, c'est faux, car il manque un lien entre l'environnement de typage Γ et l'environnement d'exécution E

contre-exemple :

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

Définition (environnement bien typé)

Un environnement d'exécution E est bien typé dans un environnement de typage Γ, noté $\Gamma \vdash E$, si

$$\forall x,\ si\ \Gamma(x) = \{\dots f : \tau \dots\}\ alors\ (x, f) \in dom(E)\ et\ \Gamma \vdash E(x, f) : \tau$$

Lemme (évaluation d'une expression bien typée)

Si $\Gamma \vdash e : \tau$ *et* $\Gamma \vdash E$*, alors* $E, e \twoheadrightarrow v$ *et* $\Gamma \vdash v : \tau$*.*

preuve : par récurrence sur la dérivation $\Gamma \vdash e : \tau$.

$e = c$ immédiat avec $v = c$

$e = x$ immédiat avec $v = x$

$e = e_1.f$ par HR $E, e_1 \twoheadrightarrow v_1$ et $\Gamma \vdash v_1 : \tau_1$ avec $\tau_1 = \{\dots f : \tau \dots\}$.
donc v_1 est une variable x et $v = E(x, f)$
comme E est bien typé, on a $\Gamma \vdash v : \tau$

$e = e_1 + e_2$ par HR sur e_1 et e_2 on a $E, e_i \twoheadrightarrow v_i$ et $\Gamma \vdash v_i : \texttt{int}$, donc v_1 et v_2 sont des entiers et on conclut avec $v = v_1 + v_2$

□

la preuve de la sûreté s'appuie sur deux lemmes

Lemme (progrès)

Si $\Gamma \vdash s$ *et* $\Gamma \vdash E$*, alors soit* s *est* `skip`*, soit* $E, s \rightarrow E', s'$*.*

Lemme (préservation)

Si $\Gamma \vdash s$*, si* $\Gamma \vdash E$ *et si* $E, s \rightarrow E', s'$ *alors* $\Gamma \vdash s'$ *et* $\Gamma \vdash E'$*.*

Lemme (progrès)

Si $\Gamma \vdash s$ *et* $\Gamma \vdash E$*, alors soit* s *est* `skip`*, soit* $E, s \rightarrow E', s'$*.*

preuve : par récurrence sur la dérivation $\Gamma \vdash s$

$s = \mathtt{skip}$ immédiat

$s = s_1; s_2$ si $s_1 = \mathtt{skip}$, on a $E, s_1; s_2 \rightarrow E, s_2$
sinon, on applique l'HR à s_1, d'où $E, s_1 \rightarrow E', s_1'$ et donc
$E, s_1; s_2 \rightarrow E', s_1'; s_2$

$s = e_1.f \leftarrow e_2$ comme e_1 et e_2 sont bien typées, elles s'évaluent en x et v respectivement
comme $\Gamma \vdash x : \{\dots f : \tau \dots\}$ alors $(x, f) \in \mathrm{dom}(E)$ et donc
$E, s \rightarrow E', \mathtt{skip}$ avec $E' = E\{(x, f) \mapsto v\}$

autres cas laissés en exercice $\square$

on montre ensuite

Lemme (préservation)

Si $\Gamma \vdash s$*, si* $\Gamma \vdash E$ *et si* $E, s \rightarrow E', s'$ *alors* $\Gamma \vdash s'$ *et* $\Gamma \vdash E'$*.*

preuve : par récurrence sur la dérivation $\Gamma \vdash s$

$s = s_1; s_2$ on a $\Gamma \vdash s_1$ et $\Gamma \vdash s_2$

- si $s_1 = \texttt{skip}$, alors $E, s_1; s_2 \rightarrow E, s_2$
- sinon, $E, s_1 \rightarrow E', s_1'$ et par HR $\Gamma \vdash s_1'$ et $\Gamma \vdash E'$ d'où $\Gamma \vdash s_1'; s_2$

$s = e_1.f \leftarrow e_2$ on a $E, e_1 \twoheadrightarrow x$, $E, e_2 \twoheadrightarrow v$, $s' = \texttt{skip}$ (donc $\Gamma \vdash s'$) et $E' = E\{(x, f) \mapsto v\}$
or $\Gamma \vdash e_1 : \{\dots f : \tau \dots\}$ et $\Gamma \vdash e_2 : \tau$ donc $\Gamma \vdash v : \tau$ (cf transparent **??**) et donc $\Gamma \vdash E'$

autres cas laissés en exercice □

on peut maintenant prouver le théorème facilement

Théorème (sûreté du typage)

Si $\Gamma \vdash s$ et $E, s \rightarrow^\star E', s'$ et s' irréductible, alors s' est `skip`.

preuve : on a $E, s \rightarrow E_1, s_1 \rightarrow \cdots \rightarrow E', s'$ et par applications répétées du lemme de préservation, on a donc $\Gamma \vdash s'$
par le lemme de progrès, s' se réduit ou est `skip`
c'est donc `skip`

le poly contient une preuve similaire pour le langage Mini-ML,
avec des types de la forme

$$
\begin{array}{rcll}
\tau & ::= & \texttt{int} \mid \texttt{bool} \mid \dots & \textit{types de base} \\
 & \mid & \tau \rightarrow \tau & \textit{type d'une fonction} \\
 & \mid & \tau \times \tau & \textit{type d'une paire}
\end{array}
$$

comme ici, la preuve repose sur les propriétés de progrès et de préservation

cf chapitre 5

des langages comme Java ou OCaml possèdent une telle propriété de sûreté du typage

ce qui signifie que l'évaluation d'une expression de type τ

- soit ne termine pas
- soit lève une exception
- soit termine sur une valeur **de type τ**

en OCaml, l'absence de `null` fait de ce dernier cas une propriété très forte

il y a une différence entre les **règles de typage**, qui définissent la relation ternaire

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

et l'**algorithme de typage** qui vérifie qu'une certaine expression e est bien typée dans un certain environnement Γ

par exemple

- le type τ pourrait ne pas être donné (inférence de types)
- plusieurs règles pourraient s'appliquer à une même expression
- en particulier, une même expression pourrait avoir plusieurs types

dans notre cas

ici c'est simple, car une seule règle s'applique à chaque fois

on dit que le typage est **dirigé par la syntaxe** (*syntax-directed*)

le typage est donc réalisé par un parcours de complexité linéaire

en pratique

- on ne se contente pas d'un message

```
erreur de typage
```

  mais on indique l'origine du problème avec précision

- on conserve les types pour les phases ultérieures du compilateur

pour cela, on **décore** les arbres de syntaxe abstraite

- **en entrée** du typage avec une localisation dans le fichier source
- **en sortie** du typage avec les types obtenus

en OCaml

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

```
type expr =




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

en Java

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

```
abstract class Expr {



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

en OCaml

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

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

en Java

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

```
abstract class Expr {
  Loc loc;


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

on signale une erreur de typage en levant une exception

cette exception contient

- la nature de l'erreur (un message précis)
- une localisation

on rattrape cette exception dans le programme principal

on affiche la localisation et le message

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

on se donne une syntaxe abstraite pour les types

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

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

et une **nouvelle** syntaxe abstraite pour les programmes

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

le typage transforme des arbres de syntaxe abstraite d'un certain type en des arbres de syntaxe abstraite d'un **autre** type

on **reconstruit** de nouveaux arbres

cela reste efficace, car

- c'est typiquement un parcours linéaire
- les anciens arbres seront récupérés par le GC

sous-typage

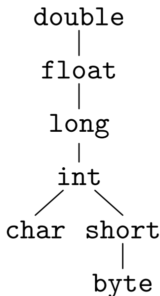
on dit qu'un type τ_1 est un **sous-type** d'un type τ_2, ce que l'on note

$$\tau_1 \leq \tau_2$$

si toute valeur de type τ_1 peut être vue comme une valeur de type τ_2

dans beaucoup de langages, il existe un sous-typage entre types numériques

en Java, ce sous-typage est le suivant :



ainsi on peut écrire

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

mais pas

```
byte b = 144;
```

dans un langage orienté objet, la notion d'héritage s'accompagne d'une notion de **sous-typage** : si une classe `B` hérite d'une classe `A`, on a

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

i.e. toute valeur de type `B` peut être vue comme une valeur de type `A`

les deux classes

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

introduisent le sous-typage

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

et on peut donc écrire

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

la construction `new C(...)` construit un objet de classe `C`, et la classe de cet objet ne peut être modifiée par la suite ; on l'appelle le **type dynamique** de l'objet

en revanche, le **type statique** d'une expression, tel qu'il est calculé par le compilateur, peut être différent du type dynamique, du fait de la relation de sous-typage introduite par l'héritage

quand on écrit

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

pour le compilateur, `v` a le type `Vehicle`,
mais c'est bien la méthode `move` de la classe `Car` qui est exécutée

dans beaucoup de circonstances, le compilateur **ne peut pas** connaître le type dynamique

exemple :

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

il est parfois nécessaire de « forcer la main » au compilateur, en prétendant qu'une valeur appartient à un certain type

on appelle cela le **transtypage** (en anglais *cast*)

la notation de Java (héritée du langage C) est

$$(\tau)e$$

le type statique d'une telle expression est τ

exemple

ainsi on peut écrire en Java

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

ici, il n'y a pas de vérification dynamique
(si l'entier est trop grand, il est tronqué)

considérons l'expression

$$(C)e$$

soit

- D le type dynamique de (l'objet désigné par) e
- E le type statique de l'expression e

il y a trois situations

- C est une super classe de E : on parle d'**upcast** et le code produit pour $(C)e$ est le même que pour e (mais le *cast* a une influence sur le typage puisque le type de $(C)e$ est C)
- C est une sous-classe de E : on parle de **downcast** et le code contient un **test dynamique** pour vérifier que D est bien une sous-classe de C
- C n'est ni une sous-classe ni une super classe de E : le compilateur refuse l'expression

```
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
```

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

rien ne garantit que l'objet passé à `m` sera bien une voiture ; en particulier il pourrait ne même pas posséder de méthode `await` !

le test dynamique est donc nécessaire

l'exception `ClassCastException` est levée si le test échoue

pour permettre une programmation un peu plus défensive, il existe une construction booléenne

$$e \ \texttt{instanceof} \ C$$

qui détermine si la classe de e est bien une sous-classe de C

on trouve souvent le schéma

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

dans ce cas, le compilateur effectue typiquement une optimisation consistant à ne pas générer de second test pour le *cast*

surcharge

la **surcharge** (en anglais *overloading*) est la capacité d'utiliser le même nom pour plusieurs opérations

la surcharge est résolue au typage, grâce au nombre et aux types (statiques) des arguments

en Java, l'opération + est surchargée

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

il faut le voir comme trois opérations

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

quand on écrit

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

c'est le sous-typage qui permet de voir `'a'` de type `char` comme une valeur de type `int`, et il s'agit donc de l'opération `+(int, int)`

mais quand on écrit

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

ce n'est **pas** du sous-typage (`int` $\not\leq$ `String`)

en particulier, on ne peut pas écrire

```
String t = 42;
```

en Java, l'utilisateur ne peut pas surcharger les opérateurs comme +
mais il peut surcharger les méthodes/constructeurs

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

tout se passe comme si on avait défini plutôt

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

et le compilateur utilise les types (statiques) des arguments d'un appel à `f` pour déterminer quelle méthode appeler

résoudre la surcharge peut être délicat

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

dans le code

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

les deux méthodes s'appliquent potentiellement

c'est la méthode `m(B b)` qui est appelée,
car **plus précise** du point de vue du type de l'argument

il peut y avoir ambiguïté

```
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); ... }
```

```
surcharge1.java:13: reference to m is ambiguous,
  both method m(A,B) in B and method m(B,A) in B match
```

à chaque méthode définie dans la classe C

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

on associe le profil $(\mathtt{C}, \tau_1, \ldots, \tau_n)$

on **ordonne** les profils : $(\tau_0, \tau_1, \ldots, \tau_n) \sqsubseteq (\tau'_0, \tau'_1, \ldots, \tau'_n)$ si et seulement si τ_i est un sous-type de τ'_i pour tout i

pour un appel

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

où e a le type statique τ_0 et e_i le type statique τ_i, on considère l'ensemble des éléments **minimaux** dans l'ensemble des profils compatibles

- aucun élément $\Rightarrow$ aucune méthode ne s'applique
- plusieurs éléments $\Rightarrow$ ambiguïté
- un unique élément $\Rightarrow$ c'est la méthode à appeler