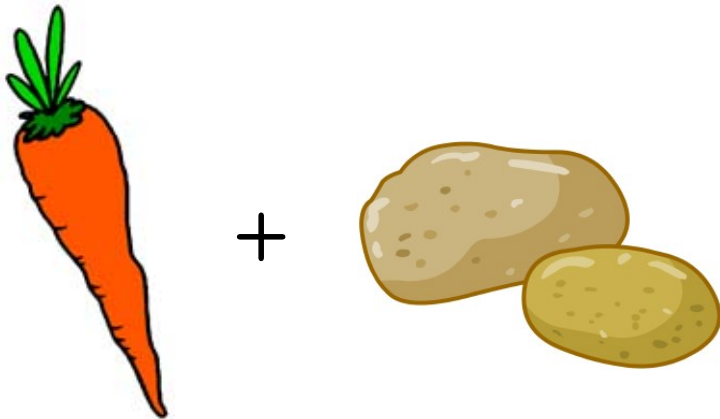


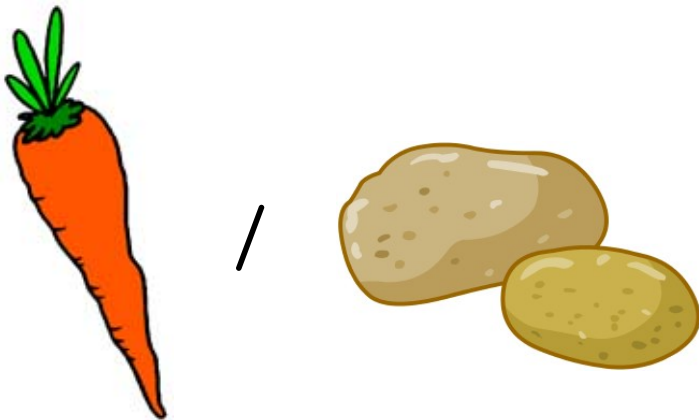
Université Paris-Sud

Compilation et langages

Thibaut Balabonski
d'après Jean-Christophe Filliâtre @ ENS

Cours 8 / 18 novembre 2016





Si j'écris l'expression

```
"5" + 37
```

Si j'écris l'expression

```
"5" + 37
```

Dois-je obtenir

- une erreur à la compilation ? (OCaml)
- une erreur à l'exécution ? (Python)
- l'entier 42 ? (Visual Basic, PHP)
- la chaîne "537" ? (Java)
- 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 les buts de :

- rejeter les programmes incohérents,
- aider la compilation de certaines constructions.

Certains langages sont **typés dynamiquement**, c'est-à-dire pendant l'exécution du programme

Exemples : Lisp, PHP, Python

D'autres sont **typés statiquement**, c'est-à-dire pendant la compilation du programme

Exemples : C, Java, OCaml

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

Well typed programs do not go wrong

- Le typage doit être **décidable**
- Le typage doit rejeter les programmes absurdes comme 1 2, 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

```
fun (x : int) → let (y : int) = (+ :)(((x : int), (1 : int))) : int × int)
```

facile à vérifier mais trop fastidieux pour le programmeur

2. Annoter seulement les déclarations de variables (Pascal, C, Java, etc.)

```
fun (x : int) → let (y : int) = +(x, 1) in y
```

3. Annoter seulement les paramètres de fonctions

```
fun (x : int) → let y = +(x, 1) in y
```

4. Ne rien annoter \Rightarrow **inférence** complète (OCaml, Haskell, etc.)

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

On considère le typage (monomorphe) de mini-ML

$e ::=$	x	identificateur
	c	constante ($1, 2, \dots, true, \dots$)
	op	primitive ($+, \times, fst, \dots$)
	$\text{fun } x \rightarrow e$	fonction
	$e \ e$	application
	(e, e)	paire
	$\text{let } x = e \text{ in } e$	liaison locale

On se donne des **types simples**, dont la syntaxe abstraite est

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

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 »

L'environnement Γ associe un type $\Gamma(x)$ à toute variable x libre dans e

On va définir la relation \vdash à l'aide de règles d'inférences (cf. cours de sémantique)

$$\overline{\Gamma \vdash x : \Gamma(x)} \quad \overline{\Gamma \vdash n : \text{int}} \cdots \quad \overline{\Gamma \vdash + : \text{int} \times \text{int} \rightarrow \text{int}} \cdots$$

$$\frac{\Gamma \vdash x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \text{fun } x \rightarrow e : \tau_1 \rightarrow \tau_2}$$

$$\frac{\Gamma \vdash e_1 : \tau' \rightarrow \tau \quad \Gamma \vdash e_2 : \tau'}{\Gamma \vdash e_1 \ e_2 : \tau}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : \tau_2}$$

$\Gamma \vdash x : \tau$ est l'environnement Γ' défini par $\Gamma'(x) = \tau$ et $\Gamma'(y) = \Gamma(y)$ sinon

$$\frac{\frac{\frac{\vdots}{x : \text{int}} \vdash (x, 1) : \text{int} \times \text{int}}{x : \text{int}} \vdash +(x, 1) : \text{int}}{\emptyset \vdash \text{fun } x \rightarrow +(x, 1) : \text{int} \rightarrow \text{int}} \quad \frac{\frac{\vdots}{\dots \vdash f : \text{int} \rightarrow \text{int}}}{} \quad \frac{\vdots}{\dots \vdash 2 : \text{int}}}{f : \text{int} \rightarrow \text{int} \vdash f \ 2 : \text{int}}{\emptyset \vdash \text{let } f = \text{fun } x \rightarrow +(x, 1) \text{ in } f \ 2 : \text{int}}$$

En revanche, on ne peut pas typer le programme 1 2

$$\frac{\Gamma \vdash 1 : \tau' \rightarrow \tau \quad \Gamma \vdash 2 : \tau'}{\Gamma \vdash 1 \ 2 : \tau}$$

Ni le programme `fun x → x x`

$$\frac{\frac{\Gamma + x : \tau_1 \vdash x : \tau_3 \rightarrow \tau_2 \quad \Gamma + x : \tau_1 \vdash x : \tau_3}{\Gamma + x : \tau_1 \vdash x \ x : \tau_2}}{\Gamma \vdash \text{fun } x \rightarrow x \ x : \tau_1 \rightarrow \tau_2}$$

Car $\tau_1 = \tau_1 \rightarrow \tau_2$ n'a pas de solution (les types sont finis)

On peut montrer

$$\emptyset \vdash \text{fun } x \rightarrow x : \text{int} \rightarrow \text{int}$$

Mais aussi

$$\emptyset \vdash \text{fun } x \rightarrow x : \text{bool} \rightarrow \text{bool}$$

Attention : ce n'est pas du polymorphisme

Pour une occurrence donnée de `fun x → x` il faut **choisir** un type

Ainsi, le terme `let $f = \text{fun } x \rightarrow x$ in (f 1, f true)` n'est pas typable

Car il n'y a pas de type τ tel que

$$f : \tau \rightarrow \tau \vdash (f \text{ 1}, f \text{ true}) : \tau_1 \times \tau_2$$

En particulier, on ne peut pas donner un type satisfaisant à une primitive comme `fst` ; il faudrait choisir entre

```
int × int → int
int × bool → int
bool × int → bool
(int → int) × int → int → int
etc.
```

De même pour les primitives *opif* et *opfix*

On ne peut pas donner de type à *opfix* de manière générale, mais on peut donner une règle de typage pour une construction `let rec` qui serait primitive

$$\frac{\Gamma + x : \tau_1 \vdash e_1 : \tau_1 \quad \Gamma + x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \text{let rec } x = e_1 \text{ in } e_2 : \tau_2}$$

Et si on souhaite exclure les *valeurs* récursives, on peut modifier ainsi

$$\frac{\Gamma + (f : \tau \rightarrow \tau_1) + (x : \tau) \vdash e_1 : \tau_1 \quad \Gamma + (f : \tau \rightarrow \tau_1) \vdash e_2 : \tau_2}{\Gamma \vdash \text{let rec } f \ x = e_1 \text{ in } e_2 : \tau_2}$$

Différence entre règles de typage et algorithme de typage

Quand on type $\text{fun } x \rightarrow e$, comment trouve-t-on le type à donner à x ?

C'est toute la 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

Considérons l'approche où seuls les paramètres de fonctions sont annotés et programmons-la en OCaml

Syntaxe abstraite des types

```
type typ =  
  | Tint  
  | Tarrow of typ * typ  
  | Tproduct of typ * typ
```

Le constructeur Fun prend un argument supplémentaire

```
type expression =  
  | Var of string  
  | Const of int  
  | Op of string  
  | Fun of string * typ * expression (* seul changement *)  
  | App of expression * expression  
  | Pair of expression * expression  
  | Let of string * expression * expression
```


L'environnement Γ est réalisé par une structure persistante

En l'occurrence on utilise le module Map d'OCaml

```
module Smap = Map.Make(String)

type env = typ Smap.t
```

(performances : arbres équilibrés \Rightarrow insertion et recherche en $O(\log n)$)

```
let rec type_expr env = function
  | Const _ -> Tint
  | Var x -> Smap.find x env
  | Op "+" -> Tarrow (Tproduct (Tint, Tint), Tint)
  | Pair (e1, e2) ->
      Tproduct (type_expr env e1, type_expr env e2)
```

Pour la fonction, le type de la variable est donné

```
| Fun (x, ty, e) ->
    Tarrow (ty, type_expr (Smap.add x ty env) e)
```

Pour la variable locale, il est calculé

```
| Let (x, e1, e2) ->
    type_expr (Smap.add x (type_expr env e1) env) e2
```

(noter l'intérêt de la persistance de env)

Les seules vérifications se trouvent dans l'application

```
| App (e1, e2) -> begin match type_expr env e1 with
  | Tarrow (ty2, ty) ->
    if type_expr env e2 = ty2 then ty
    else failwith "mal typé : argument de mauvais type"
  | _ ->
    failwith "mal typé : fonction attendue"
end
```

Exemples

```
# type_expr
  (Let ("f",
    Fun ("x", Tint, App (Op "+", Pair (Var "x", Const 1))),
    App (Var "f", Const 2))));;
```

```
- : typ = Tint
```

```
# type_expr (Fun ("x", Tint, App (Var "x", Var "x")));;
```

```
Exception: Failure "mal typé : fonction attendue".
```

```
# type_expr (App (App (Op "+", Const 1), Const 2));;
```

```
Exception: Failure "mal typé : argument de mauvais type".
```

- On ne fait pas

```
failwith "erreur de typage"
```

mais on indique l'origine du problème avec précision

- On conserve les types pour les phases ultérieures du compilateur

D'une part on décore les arbres **en entrée** du typage avec une localisation dans le fichier source

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

```
type expression =
```

```
| Var    of string
| Const  of int
| Op     of string
| Fun    of string * typ * expression
| App    of expression * expression
| Pair   of expression * expression
| Let    of string * expression * expression
```

D'une part on décore les arbres **en entrée** du typage avec une localisation dans le fichier source

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

```
type expression = {  
  desc: desc;  
  loc : loc;  
}
```

```
and desc =  
  | Var    of string  
  | Const  of int  
  | Op     of string  
  | Fun    of string * typ * expression  
  | App    of expression * expression  
  | Pair   of expression * expression  
  | Let    of string * expression * expression
```

On déclare une exception de la forme

```
exception Error of loc * string
```

On la lève ainsi

```
let rec type_expr env e = match e.desc with  
  | ...  
  | App (e1, e2) -> begin match type_expr env e1 with  
    | Tarrow (ty2, ty) ->  
      if type_expr env e2 <> ty2 then  
        raise (Error (e2.loc, "argument de mauvais type"));  
      ...
```


Et on la rattrape ainsi, par exemple dans le programme principal

```
try
  let p = Parser.parse file in
  let t = Typing.program p in
  ...
with Error (loc, msg) ->
  Format.eprintf "File '%s', line ...\n" file loc;
  Format.eprintf "error: %s@." msg;
  exit 1
```

D'autre part on décore les arbres **en sortie** du typage avec des types

```
type texpression = {  
  tdesc: tdesc;  
  typ  : typ;  
}  
and tdesc =  
  | Tvar    of string  
  | Tconst  of int  
  | Top     of string  
  | Tfun    of string * typ * texpression  
  | Tapp    of texpression * texpression  
  | Tpair   of texpression * texpression  
  | Tlet    of string * texpression * texpression
```

C'est un **autre type** d'expressions

La fonction de typage a donc un type de la forme

```
val type_expr: expression -> texpression
```

La fonction de typage **reconstruit** des arbres, cette fois typés

```
let rec type_expr env e =  
  let d, ty = compute_type env e in  
  { tdesc = d; typ = ty }  
  
and compute_type env e = match e.desc with  
| Const n ->  
  Tconst n, Tint  
| Var x ->  
  Tvar x, Smap.find x env  
| Pair (e1, e2) ->  
  let te1 = type_expr env e1 in  
  let te2 = type_expr env e2 in  
  Tpair (te1, te2), Tproduct (te1.typ, te2.typ)  
| ...
```

Well typed programs do not go wrong

On montre l'adéquation du typage par rapport à la sémantique à réductions (sémantique à petits pas)

Théorème (sûreté du typage)

Si $\emptyset \vdash e : \tau$, alors la réduction de e est infinie ou se termine sur une valeur.

Ou, de manière équivalente,

Théorème

Si $\emptyset \vdash e : \tau$ et $e \xrightarrow{} e'$ et e' irréductible, alors e' est une valeur.*

La preuve de ce théorème s'appuie sur deux lemmes

Lemme (progression)

Si $\emptyset \vdash e : \tau$, alors soit e est une valeur, soit il existe e' tel que $e \rightarrow e'$.

Lemme (préservation)

Si $\emptyset \vdash e : \tau$ et $e \rightarrow e'$ alors $\emptyset \vdash e' : \tau$.

Lemme (progression)

Si $\emptyset \vdash e : \tau$, alors soit e est une valeur, soit il existe e' tel que $e \rightarrow e'$.

Preuve : par récurrence sur la dérivation $\emptyset \vdash e : \tau$

Supposons par exemple $e = e_1 \ e_2$; on a donc

$$\frac{\emptyset \vdash e_1 : \tau' \rightarrow \tau \quad \emptyset \vdash e_2 : \tau'}{\emptyset \vdash e_1 \ e_2 : \tau}$$

On applique l'HR à e_1

- si $e_1 \rightarrow e'_1$, alors $e_1 \ e_2 \rightarrow e'_1 \ e_2$ (cf lemme passage au contexte)
- si e_1 est une valeur, supposons $e_1 = \text{fun } x \rightarrow e_3$ (il y a aussi $+$ etc.)
on applique l'HR à e_2
 - si $e_2 \rightarrow e'_2$, alors $e_1 \ e_2 \rightarrow e_1 \ e'_2$ (même lemme)
 - si e_2 est une valeur, alors $e_1 \ e_2 \rightarrow e_3[x \leftarrow e_2]$

(exercice : traiter les autres cas)

□

On commence par de petits lemmes faciles

Lemme (permutation)

Si $\Gamma + x : \tau_1 + y : \tau_2 \vdash e : \tau$ et $x \neq y$ alors $\Gamma + y : \tau_2 + x : \tau_1 \vdash e : \tau$ (et la dérivation a la même hauteur).

Preuve : récurrence immédiate



Lemme (affaiblissement)

Si $\Gamma \vdash e : \tau$ et $x \notin \text{dom}(\Gamma)$, alors $\Gamma + x : \tau' \vdash e : \tau$ (et la dérivation a la même hauteur).

Preuve : récurrence immédiate



On continue par un **lemme clé**

Lemme (préservation par substitution)

Si $\Gamma + x : \tau' \vdash e : \tau$ et $\Gamma \vdash e' : \tau'$ alors $\Gamma \vdash e[x \leftarrow e'] : \tau$.

Preuve : par récurrence sur la dérivation $\Gamma + x : \tau' \vdash e : \tau$

- cas d'une variable $e = y$
 - si $x = y$ alors $e[x \leftarrow e'] = e'$ et $\tau = \tau'$
 - si $x \neq y$ alors $e[x \leftarrow e'] = e$ et $\tau = \Gamma(y)$
- cas d'une abstraction $e = \text{fun } y \rightarrow e_1$
 on peut supposer $y \neq x$, $y \notin \text{dom}(\Gamma)$ et y non libre dans e' (α -conversion)
 on a $\Gamma + x : \tau' + y : \tau_2 \vdash e_1 : \tau_1$ et donc $\Gamma + y : \tau_2 + x : \tau' \vdash e_1 : \tau_1$
 (permutation); d'autre part $\Gamma \vdash e' : \tau'$ et donc $\Gamma + y : \tau_2 \vdash e' : \tau'$
 (affaiblissement)
 par HR on a donc $\Gamma + y : \tau_2 \vdash e_1[x \leftarrow e'] : \tau_1$
 et donc $\Gamma \vdash (\text{fun } y \rightarrow e_1)[x \leftarrow e'] : \tau_2 \rightarrow \tau_1$, i.e. $\Gamma \vdash e[x \leftarrow e'] : \tau$

(exercice : traiter les autres cas)

□

On peut enfin montrer

Lemme (préservation)

Si $\emptyset \vdash e : \tau$ et $e \rightarrow e'$ alors $\emptyset \vdash e' : \tau$.

Preuve : par récurrence sur la dérivation $\emptyset \vdash e : \tau$

- cas $e = \text{let } x = e_1 \text{ in } e_2$

$$\frac{\emptyset \vdash e_1 : \tau_1 \quad x : \tau_1 \vdash e_2 : \tau_2}{\emptyset \vdash \text{let } x = e_1 \text{ in } e_2 : \tau_2}$$

- si $e_1 \rightarrow e'_1$, par HR on a $\emptyset \vdash e'_1 : \tau_1$ et donc $\emptyset \vdash \text{let } x = e'_1 \text{ in } e_2 : \tau_2$
- si e_1 est une valeur et $e' = e_2[x \leftarrow e_1]$ alors on applique le lemme de préservation par substitution
- cas $e = e_1 \ e_2$
 - si $e_1 \rightarrow e'_1$ ou si e_1 valeur et $e_2 \rightarrow e'_2$ alors on utilise l'HR
 - si $e_1 = \text{fun } x \rightarrow e_3$ et e_2 valeur alors $e' = e_3[x \leftarrow e_2]$ et on applique là encore le lemme de substitution □

On peut maintenant prouver le théorème facilement

Théorème (sûreté du typage)

Si $\emptyset \vdash e : \tau$ et $e \xrightarrow{} e'$ et e' irréductible, alors e' est une valeur.*

Preuve : on a $e \rightarrow e_1 \rightarrow \dots \rightarrow e'$ et par applications répétées du lemme de préservation, on a donc $\emptyset \vdash e' : \tau$
par le lemme de progression, e' se réduit ou est une valeur
c'est donc une valeur □

Il est contraignant de donner un type unique à `fun x → x` dans

```
let f = fun x → x in ...
```

De même, on aimerait pouvoir donner « plusieurs types » aux primitives telles que `fst` ou `snd`

Une solution : le **polymorphisme paramétrique**

On étend l'algèbre des types :

$\tau ::=$	$\text{int} \mid \text{bool} \mid \dots$	<i>types de base</i>
	$\mid \tau \rightarrow \tau$	<i>type d'une fonction</i>
	$\mid \tau \times \tau$	<i>type d'une paire</i>
	$\mid \alpha$	<i>variable de type</i>
	$\mid \forall \alpha. \tau$	<i>type polymorphe</i>

Les règles sont **exactement** les mêmes qu'auparavant, plus

$$\frac{\Gamma \vdash e : \tau \quad \alpha \notin \mathcal{L}(\Gamma)}{\Gamma \vdash e : \forall \alpha. \tau}$$

et

$$\frac{\Gamma \vdash e : \forall \alpha. \tau}{\Gamma \vdash e : \tau[\alpha \leftarrow \tau']}$$

Le système obtenu s'appelle le **système F** (J.-Y. Girard / J. C. Reynolds)

Dans la règle

$$\frac{\Gamma \vdash e : \tau \quad \alpha \notin \mathcal{L}(\Gamma)}{\Gamma \vdash e : \forall \alpha. \tau}$$

$\mathcal{L}(\Gamma)$ dénote l'ensemble des variables de types **libres** dans Γ , défini par

$$\mathcal{L}(\Gamma) = \bigcup_{x \in \text{dom}(\Gamma)} \mathcal{L}(\Gamma(x))$$

$$\mathcal{L}(\text{int}) = \emptyset$$

$$\mathcal{L}(\alpha) = \{\alpha\}$$

$$\mathcal{L}(\tau_1 \rightarrow \tau_2) = \mathcal{L}(\tau_1) \cup \mathcal{L}(\tau_2)$$

$$\mathcal{L}(\tau_1 \times \tau_2) = \mathcal{L}(\tau_1) \cup \mathcal{L}(\tau_2)$$

$$\mathcal{L}(\forall \alpha. \tau) = \mathcal{L}(\tau) \setminus \{\alpha\}$$

$$\frac{
 \frac{
 \frac{x : \alpha \vdash x : \alpha}{\vdash \text{fun } x \rightarrow x : \alpha \rightarrow \alpha}
 }{
 \vdash \text{fun } x \rightarrow x : \forall \alpha. \alpha \rightarrow \alpha
 }
 \quad
 \frac{
 \frac{
 \frac{\dots \vdash f : \forall \alpha. \alpha \rightarrow \alpha}{\dots \vdash f : \text{int} \rightarrow \text{int}}
 \quad \vdots
 }{
 \dots \vdash f \text{ 1} : \text{int}
 }
 \quad
 \frac{
 \vdots
 }{
 \dots \vdash f \text{ true} : \text{bool}
 }
 }{
 f : \forall \alpha. \alpha \rightarrow \alpha \vdash (f \text{ 1}, f \text{ true}) : \text{int} \times \text{bool}
 }
 }{
 \vdash \text{let } f = \text{fun } x \rightarrow x \text{ in } (f \text{ 1}, f \text{ true}) : \text{int} \times \text{bool}
 }$$

On peut maintenant donner un type satisfaisant aux primitives

$$fst : \forall \alpha. \forall \beta. \alpha \times \beta \rightarrow \alpha$$

$$snd : \forall \alpha. \forall \beta. \alpha \times \beta \rightarrow \beta$$

$$opif : \forall \alpha. \text{bool} \times \alpha \times \alpha \rightarrow \alpha$$

$$opfix : \forall \alpha. (\alpha \rightarrow \alpha) \rightarrow \alpha$$

On peut construire une dérivation de

$$\Gamma \vdash \text{fun } x \rightarrow x \ x : (\forall \alpha. \alpha \rightarrow \alpha) \rightarrow (\forall \alpha. \alpha \rightarrow \alpha)$$

(exercice : le faire)

La condition $\alpha \notin \mathcal{L}(\Gamma)$ dans la règle

$$\frac{\Gamma \vdash e : \tau \quad \alpha \notin \mathcal{L}(\Gamma)}{\Gamma \vdash e : \forall \alpha. \tau}$$

est cruciale

Sans elle, on aurait

$$\frac{\frac{\Gamma + x : \alpha \vdash x : \alpha}{\Gamma + x : \alpha \vdash x : \forall \alpha. \alpha}}{\Gamma \vdash \mathbf{fun} \ x \rightarrow x : \alpha \rightarrow \forall \alpha. \alpha}$$

et on accepterait donc le programme

$(\mathbf{fun} \ x \rightarrow x) \ 1 \ 2$

Pour des termes sans annotations, les deux problèmes

- **inférence** : étant donné e , existe-t-il τ tel que $\vdash e : \tau$?
- **vérification** : étant donnés e et τ , a-t-on $\vdash e : \tau$?

ne sont pas décidables

J. B. Wells. *Typability and type checking in the second-order lambda-calculus are equivalent and undecidable*, 1994.

Pour obtenir une inférence de types décidable, on va restreindre la puissance du système F

⇒ le système de **Hindley-Milner**, utilisé dans OCaml, SML, Haskell, ...

On limite la quantification \forall en tête des types (quantification prénexe)

$$\begin{array}{ll} \tau ::= \text{int} \mid \text{bool} \mid \dots & \text{types de base} \\ | \tau \rightarrow \tau & \text{type d'une fonction} \\ | \tau \times \tau & \text{type d'une paire} \\ | \alpha & \text{variable de type} \end{array}$$

$$\begin{array}{ll} \sigma ::= \tau & \text{schémas} \\ | \forall \alpha. \sigma & \end{array}$$

L'environnement Γ associe un schéma de type à chaque identificateur et la relation de typage a maintenant la forme $\Gamma \vdash e : \sigma$

Dans Hindley-Milner, les types suivants sont toujours acceptés

$$\forall \alpha. \alpha \rightarrow \alpha$$

$$\forall \alpha. \forall \beta. \alpha \times \beta \rightarrow \alpha$$

$$\forall \alpha. \text{bool} \times \alpha \times \alpha \rightarrow \alpha$$

$$\forall \alpha. (\alpha \rightarrow \alpha) \rightarrow \alpha$$

mais plus les types tels que

$$(\forall \alpha. \alpha \rightarrow \alpha) \rightarrow (\forall \alpha. \alpha \rightarrow \alpha)$$

Note : dans la syntaxe d'OCaml, la quantification prénexe est implicite

```
# fst;;
```

```
- : 'a * 'b -> 'a = <fun>
```

$$\forall \alpha. \forall \beta. \alpha \times \beta \rightarrow \alpha$$

```
# List.fold_left;;
```

```
- : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a = <fun>
```

$$\forall \alpha. \forall \beta. (\alpha \rightarrow \beta \rightarrow \alpha) \rightarrow \alpha \rightarrow \beta \text{ list} \rightarrow \alpha$$

$$\overline{\Gamma \vdash x : \Gamma(x)} \quad \overline{\Gamma \vdash n : \text{int}} \cdots \quad \overline{\Gamma \vdash + : \text{int} \times \text{int} \rightarrow \text{int}} \cdots$$

$$\frac{\Gamma \vdash x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \text{fun } x \rightarrow e : \tau_1 \rightarrow \tau_2}$$

$$\frac{\Gamma \vdash e_1 : \tau' \rightarrow \tau \quad \Gamma \vdash e_2 : \tau'}{\Gamma \vdash e_1 \ e_2 : \tau}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2}$$

$$\frac{\Gamma \vdash e_1 : \sigma_1 \quad \Gamma \vdash x : \sigma_1 \vdash e_2 : \sigma_2}{\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : \sigma_2}$$

$$\frac{\Gamma \vdash e : \sigma \quad \alpha \notin \mathcal{L}(\Gamma)}{\Gamma \vdash e : \forall \alpha. \sigma}$$

$$\frac{\Gamma \vdash e : \forall \alpha. \sigma}{\Gamma \vdash e : \sigma[\alpha \leftarrow \tau]}$$

On note que seule la construction `let` permet d'introduire un type polymorphe dans l'environnement

$$\frac{\Gamma \vdash e_1 : \sigma_1 \quad \Gamma + x : \sigma_1 \vdash e_2 : \sigma_2}{\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : \sigma_2}$$

En particulier, on peut toujours typer

`let $f = \text{fun } x \rightarrow x$ in (f 1, f true)`

avec $f : \forall \alpha. \alpha \rightarrow \alpha$ dans le contexte pour typer `(f 1, f true)`

En revanche, la règle de typage

$$\frac{\Gamma + x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \text{fun } x \rightarrow e : \tau_1 \rightarrow \tau_2}$$

n'introduit pas un type polymorphe (sinon $\tau_1 \rightarrow \tau_2$ serait mal formé)

En particulier, on ne peut plus typer

`fun x → x x`

Pour les références, on peut naïvement penser qu'il suffit d'ajouter les primitives

$$\begin{aligned} \text{ref} &: \forall \alpha. \alpha \rightarrow \alpha \text{ ref} \\ ! &: \forall \alpha. \alpha \text{ ref} \rightarrow \alpha \\ := &: \forall \alpha. \alpha \text{ ref} \rightarrow \alpha \rightarrow \text{unit} \end{aligned}$$

Malheureusement, c'est incorrect !

```
let r = ref (fun x → x) in
let _ = r := (fun x → x + 1) in
!r true
```

$r : \forall \alpha. (\alpha \rightarrow \alpha) \text{ ref}$

boum !

C'est le problème dit des **références polymorphes**

Pour contourner ce problème, il existe une solution extrêmement simple, à savoir une restriction syntaxique de la construction `let`

Définition (*value restriction*, Wright 1995)

Un programme satisfait le critère de la **value restriction** si toute sous-expression `let` est de la forme

$$\text{let } x = v_1 \text{ in } e_2$$

où v_1 est une valeur.

On ne peut plus écrire

```
let  $r = \text{ref } (\text{fun } x \rightarrow x)$  in ...
```

Mais on peut écrire en revanche

```
(fun  $r \rightarrow \dots$ ) (ref (fun  $x \rightarrow x$ ))
```

où le type de r n'est pas généralisé

En pratique, on continue d'écrire `let r = ref ... in ...`
mais le type de `r` n'est pas généralisé

En OCaml, une variable non généralisée est de la forme `'_a`

```
# let x = ref (fun x -> x);;
```

```
val x : ('_a -> '_a) ref
```

La *value restriction* est également légèrement relâchée pour autoriser des expressions sûres, telles que des applications de constructeurs

```
# let l = [fun x -> x];;
```

```
val l : ('a -> 'a) list = [<fun>]
```

Il reste quelques petits désagréments

```
# let id x = x;;
```

```
val id : 'a -> 'a = <fun>
```

```
# let f = id id;;
```

```
val f : '_a -> '_a = <fun>
```

```
# f 1;;
```

```
- : int = 1
```

```
# f true;;
```

This expression has type bool but is here used with type int

```
# f;;
```

```
- : int -> int = <fun>
```

La solution : expanser pour faire apparaître une fonction, *i.e.*, une valeur

```
# let f x = id id x;;
```

```
val f : 'a -> 'a = <fun>
```

(on parle d' η -expansion)

En présence du système de modules, la réalité est plus complexe encore

Étant donné un module M

```
module M : sig
  type 'a t
  val create : int -> 'a t
end
```

ai-je le droit de généraliser le type de M.create 17 ?

La réponse dépend du type 'a t : non pour une table de hachage, oui pour une liste pure, etc.

En OCaml, une indication de **variance** permet de distinguer les deux

```
type +'a t    (* on peut généraliser *)
type 'a u     (* on ne peut pas *)
```

Lire *Relaxing the value restriction*, J. Garrigue, 2004

Deux ouvrages en rapport avec ce cours

- Benjamin C. Pierce, *Types and Programming Languages*
- Xavier Leroy et Pierre Weis, *Le langage Caml*
(le dernier chapitre explique l'inférence)