École Normale Supérieure

Langages de programmation et compilation

Jean-Christophe Filliâtre

typage

si j'écris l'expression

```
"5" + 37
```

dois-je obtenir

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

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 valeur/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 le cours d'aujourd'hui

remarque

un langage peut utiliser **à la fois** des notions de typage statique et de typage dynamique

on le verra avec l'exemple de Java dans un autre cours *well typed programs cannot 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

   $$\texttt{fun}\,(x:\texttt{int})\rightarrow\texttt{let}\,(y:\texttt{int})=(+:)(((x:\texttt{int}),(1:\texttt{int})):\texttt{int}\times\texttt{int})$$

   facile de vérifier mais trop fastidieux pour le programmeur

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

   $$\texttt{fun}\ (x:\texttt{int})\rightarrow\texttt{let}\ (y:\texttt{int})=+(x,1)\ \texttt{in}\ y$$

3. annoter seulement les paramètres de fonctions

   $$\texttt{fun}\ (x:\texttt{int})\rightarrow\texttt{let}\ y=+(x,1)\ \texttt{in}\ y$$

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 considérons le typage de mini-ML

1. typage monomorphe
2. typage polymorphe, inférence de types

rappel

$$
\begin{array}{rcll}
e & ::= & x & \text{identificateur} \\
& | & c & \text{constante } (1,\ 2,\ \ldots,\ \mathit{true},\ \ldots) \\
& | & op & \text{primitive } (+,\ \times,\ \mathit{fst},\ \ldots) \\
& | & \texttt{fun}\ x \rightarrow e & \text{fonction} \\
& | & e\ e & \text{application} \\
& | & (e, e) & \text{paire} \\
& | & \texttt{let}\ x = e\ \texttt{in}\ e & \text{liaison locale}
\end{array}
$$

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

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

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

$$\frac{}{\Gamma \vdash x : \Gamma(x)} \qquad \frac{}{\Gamma \vdash n : \texttt{int}} \dots \qquad \frac{}{\Gamma \vdash + : \texttt{int} \times \texttt{int} \rightarrow \texttt{int}} \dots$$

$$\frac{\Gamma + x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \texttt{fun}\ x \rightarrow e : \tau_1 \rightarrow \tau_2} \qquad \frac{\Gamma \vdash e_1 : \tau' \rightarrow \tau \quad \Gamma \vdash e_2 : \tau'}{\Gamma \vdash e_1\ e_2 : \tau}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma + x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau_2}$$

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

$$
\dfrac{
\dfrac{
\dfrac{\vdots \quad \dfrac{\vdots}{x : \texttt{int} \vdash (x,1) : \texttt{int} \times \texttt{int}}}
{x : \texttt{int} \vdash +(x,1) : \texttt{int}}
}{\emptyset \vdash \texttt{fun}\ x \to +(x,1) : \texttt{int} \to \texttt{int}}
\qquad
\dfrac{\dfrac{}{\ldots \vdash f : \texttt{int} \to \texttt{int}} \quad \dfrac{}{\ldots \vdash 2 : \texttt{int}}}
{f : \texttt{int} \to \texttt{int} \vdash f\ 2 : \texttt{int}}
}{\emptyset \vdash \texttt{let}\ f = \texttt{fun}\ x \to +(x,1)\ \texttt{in}\ f\ 2 : \texttt{int}}
$$

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

$$\frac{\Gamma \vdash 1 : \tau' \to \tau \qquad \Gamma \vdash 2 : \tau'}{\Gamma \vdash 1\ 2 : \tau}$$

ni le programme $\texttt{fun}\ x \to x\ x$

$$\frac{\dfrac{\Gamma + x : \tau_1 \vdash x : \tau_3 \to \tau_2 \qquad \Gamma + x : \tau_1 \vdash x : \tau_3}{\Gamma + x : \tau_1 \vdash x\ x : \tau_2}}{\Gamma \vdash \texttt{fun}\ x \to x\ x : \tau_1 \to \tau_2}$$

car $\tau_1 = \tau_1 \to \tau_2$ n'a pas de solution (les types sont finis) on peut montrer

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

mais aussi

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

attention : ce n'est pas du polymorphisme

pour une occurrence donnée de $\texttt{fun}\ x \rightarrow x$ il faut **choisir** un type ainsi, le terme $\texttt{let}\ f = \texttt{fun}\ x \rightarrow x\ \texttt{in}\ (f\ 1, f\ \texttt{true})$ n'est pas typable

car il n'y a pas de type τ tel que

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

en revanche,

$$((\texttt{fun}\ x \rightarrow x)\ (\texttt{fun}\ x \rightarrow x))\ 42$$

est typable (exercice : le montrer)

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

$$
\begin{array}{l}
\mathtt{int} \times \mathtt{int} \rightarrow \mathtt{int} \\
\mathtt{int} \times \mathtt{bool} \rightarrow \mathtt{int} \\
\mathtt{bool} \times \mathtt{int} \rightarrow \mathtt{bool} \\
(\mathtt{int} \rightarrow \mathtt{int}) \times \mathtt{int} \rightarrow \mathtt{int} \rightarrow \mathtt{int} \\
\text{etc.}
\end{array}
$$

mais on peut en revanche donner **une règle** de typage pour l'application de `fst` :

$$
\frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \mathtt{fst}\ e : \tau_1}
$$

de même, on ne peut pas donner de type à *opfix* de manière générale, mais on peut donner une règle de typage pour son application

$$\frac{\Gamma \vdash e : \tau \rightarrow \tau}{\Gamma \vdash \mathtt{opfix}\ e : \tau}$$

et si on souhaite se limiter à des *fonctions*, on peut modifier ainsi

$$\frac{\Gamma \vdash e : (\tau_1 \rightarrow \tau_2) \rightarrow (\tau_1 \rightarrow \tau_2)}{\Gamma \vdash \mathtt{opfix}\ e : \tau_1 \rightarrow \tau_2}$$

si on ajoutait plutôt une construction `let rec` dans le langage, cela donnerait

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

ou encore, pour des fonctions uniquement,

$$\frac{\Gamma + (f : \tau \to \tau_1) + (x : \tau) \vdash e_1 : \tau_1 \qquad \Gamma + (f : \tau \to \tau_1) \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{let rec } f\ x = e_1 \texttt{ in } e_2 : \tau_2}$$

différence entre règles de typage et algorithme de typage

quand on type `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

on se donne une 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 "erreur : argument de mauvais type"
      | _ ->
          failwith "erreur : 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 "erreur : fonction attendue".
```

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

```
Exception: Failure "erreur : 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 cannot go wrong

on montre l'adéquation du typage par rapport à la sémantique à réductions

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* $e \xrightarrow{\star} e'$ *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* $e \to e'$.

Lemme (préservation)

Si $\emptyset \vdash e : \tau$ *et* $e \to e'$ *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 $e_1 = \texttt{fun}\ x \rightarrow e_3$ (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 $\square$

Lemme (affaiblissement)

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

preuve : récurrence immédiate $\square$

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 (hauteur de 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 $e = \texttt{fun}\ y \rightarrow e_1$

  on peut supposer $y \neq x$, $y \notin dom(\Gamma)$ 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
  $\Gamma \vdash (\texttt{fun}\ y \rightarrow e_1)[x \leftarrow e'] : \tau_2 \rightarrow \tau_1$, *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 $e = \texttt{let } x = e_1 \texttt{ in } e_2$

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

  - si $e_1 \rightarrow e_1'$, par HR on a $\emptyset \vdash e_1' : \tau_1$ et donc $\emptyset \vdash \texttt{let } x = e_1' \texttt{ in } e_2 : \tau_2$
  - 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 $e_1 = \texttt{fun } x \rightarrow e_3$ et e_2 valeur alors $e' = e_3[x \leftarrow e_2]$ et on applique là encore le lemme de substitution $\square$

on peut maintenant prouver le théorème facilement

Théorème (sûreté du typage)

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

preuve : on a $e \rightarrow e_1 \rightarrow \cdots \rightarrow e'$ 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 $\square$

il est contraignant de donner un type unique à `fun` $x \rightarrow x$ dans

$$\texttt{let}\ f = \texttt{fun}\ x \rightarrow x\ \texttt{in}\ \ldots$$

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 :

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

on note $\mathcal{L}(\tau)$ l'ensemble des variables de types **libres** dans τ, défini par

$$
\begin{array}{rcl}
\mathcal{L}(\mathtt{int}) & = & \emptyset \\
\mathcal{L}(\alpha) & = & \{\alpha\} \\
\mathcal{L}(\tau_1 \rightarrow \tau_2) & = & \mathcal{L}(\tau_1) \cup \mathcal{L}(\tau_2) \\
\mathcal{L}(\tau_1 \times \tau_2) & = & \mathcal{L}(\tau_1) \cup \mathcal{L}(\tau_2) \\
\mathcal{L}(\forall\alpha.\tau) & = & \mathcal{L}(\tau) \setminus \{\alpha\}
\end{array}
$$

pour un environnement Γ, on définit aussi

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

on note $\tau[\alpha \leftarrow \tau']$ la substitution de α par τ' dans τ, définie par

$$
\begin{array}{rcl}
\mathtt{int}[\alpha \leftarrow \tau'] & = & \mathtt{int} \\
\alpha[\alpha \leftarrow \tau'] & = & \tau' \\
\beta[\alpha \leftarrow \tau'] & = & \beta \quad \text{si } \beta \neq \alpha \\
(\tau_1 \rightarrow \tau_2)[\alpha \leftarrow \tau'] & = & \tau_1[\alpha \leftarrow \tau'] \rightarrow \tau_2[\alpha \leftarrow \tau'] \\
(\tau_1 \times \tau_2)[\alpha \leftarrow \tau'] & = & \tau_1[\alpha \leftarrow \tau'] \times \tau_2[\alpha \leftarrow \tau'] \\
(\forall \alpha.\tau)[\alpha \leftarrow \tau'] & = & \forall \alpha.\tau \\
(\forall \beta.\tau)[\alpha \leftarrow \tau'] & = & \forall \beta.\tau[\alpha \leftarrow \tau'] \quad \text{si } \beta \neq \alpha
\end{array}
$$

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

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

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)

$$
\dfrac{
\dfrac{\dfrac{x : \alpha \vdash x : \alpha}{\vdash \mathtt{fun}\ x \to x : \alpha \to \alpha}}{\vdash \mathtt{fun}\ x \to x : \forall\alpha.\alpha \to \alpha}
\qquad
\dfrac{
\dfrac{\dfrac{\ldots \vdash f : \forall\alpha.\alpha \to \alpha}{\ldots \vdash f : \mathtt{int} \to \mathtt{int}} \quad \vdots}{\ldots \vdash f\ 1 : \mathtt{int}}
\qquad
\dfrac{\vdots}{\ldots \vdash f\ \mathtt{true} : \mathtt{bool}}
}{f : \forall\alpha.\alpha \to \alpha \vdash (f\ 1, f\ \mathtt{true}) : \mathtt{int} \times \mathtt{bool}}
}{\vdash \mathtt{let}\ f = \mathtt{fun}\ x \to x\ \mathtt{in}\ (f\ 1, f\ \mathtt{true}) : \mathtt{int} \times \mathtt{bool}}
$$

on peut maintenant donner un type satisfaisant aux primitives

$$
\begin{aligned}
\mathit{fst} &: \quad \forall\alpha.\forall\beta.\alpha \times \beta \rightarrow \alpha \\
\mathit{snd} &: \quad \forall\alpha.\forall\beta.\alpha \times \beta \rightarrow \beta \\
\mathit{opif} &: \quad \forall\alpha.\texttt{bool} \times \alpha \times \alpha \rightarrow \alpha \\
\mathit{opfix} &: \quad \forall\alpha.(\alpha \rightarrow \alpha) \rightarrow \alpha
\end{aligned}
$$

on peut construire une dérivation de

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

(exercice : le faire)

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

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

est cruciale

sans elle, on aurait

$$\dfrac{\dfrac{\dfrac{\Gamma + x : \alpha \vdash x : \alpha}{\Gamma + x : \alpha \vdash x : \forall\alpha.\alpha}}{\Gamma \vdash \texttt{fun}\ x \rightarrow x : \alpha \rightarrow \forall\alpha.\alpha}}{\Gamma \vdash \texttt{fun}\ x \rightarrow x : \forall\alpha.\alpha \rightarrow \forall\alpha.\alpha}$$

et on accepterait notamment le programme

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

pour des termes sans annotations, les deux problèmes

- **inférence** : étant donnée 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

$\Rightarrow$ le système de **Hindley-Milner**, utilisé dans OCaml, SML, Haskell, ...

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

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

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

$$\begin{array}{l} \forall\alpha.\alpha \rightarrow \alpha \\ \forall\alpha.\forall\beta.\alpha \times \beta \rightarrow \alpha \\ \forall\alpha.\texttt{bool} \times \alpha \times \alpha \rightarrow \alpha \\ \forall\alpha.(\alpha \rightarrow \alpha) \rightarrow \alpha \end{array}$$

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>
```

$$\forall\alpha.\forall\beta.\alpha \times \beta \to \alpha$$

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

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

$$\forall\alpha.\forall\beta.(\alpha \to \beta \to \alpha) \to \alpha \to \beta\ \texttt{list} \to \alpha$$

$$\frac{}{\Gamma \vdash x : \Gamma(x)} \qquad \frac{}{\Gamma \vdash n : \mathtt{int}} \dots \qquad \frac{}{\Gamma \vdash + : \mathtt{int} \times \mathtt{int} \rightarrow \mathtt{int}} \dots$$

$$\frac{\Gamma + x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \mathtt{fun}\ x \rightarrow e : \tau_1 \rightarrow \tau_2} \qquad \frac{\Gamma \vdash e_1 : \tau' \rightarrow \tau \quad \Gamma \vdash e_2 : \tau'}{\Gamma \vdash e_1\ e_2 : \tau}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2} \qquad \frac{\Gamma \vdash e_1 : \sigma_1 \quad \Gamma + x : \sigma_1 \vdash e_2 : \sigma_2}{\Gamma \vdash \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 : \sigma_2}$$

$$\frac{\Gamma \vdash e : \sigma \qquad \alpha \notin \mathcal{L}(\Gamma)}{\Gamma \vdash e : \forall \alpha . \sigma} \qquad \frac{\Gamma \vdash e : \forall \alpha . \sigma}{\Gamma \vdash e : \sigma[\alpha \leftarrow \tau]}$$

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

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

en particulier, on peut toujours typer

$$\texttt{let}\ f = \texttt{fun}\ x \to x\ \texttt{in}\ (f\ 1, f\ \texttt{true})$$

avec $f : \forall\alpha.\alpha \to \alpha$ dans le contexte pour typer $(f\ 1, f\ \texttt{true})$

en revanche, la règle de typage

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

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

en particulier, on ne peut plus typer

$$\texttt{fun}\ x \rightarrow x\ x$$

pour programmer une vérification ou une inférence de type, on procède par récurrence sur la structure du programme

or, pour une expression donnée, trois règles peuvent s'appliquer : la règle du système monomorphe, la règle

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

ou encore la règle

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

comment choisir ? va-t-on devoir procéder par essais/erreurs ?

système de Hindley-Milner dirigé par la syntaxe

nous allons modifier la présentation du système de Hindley-Milner pour qu'il soit **dirigé par la syntaxe** (*syntax-directed*), *i.e.*, pour toute expression, au plus une règle s'applique

les règles ont la même puissance d'expression : tout terme clos est typable dans un système si et seulement si il est typable dans l'autre

$$\frac{\tau \leq \Gamma(x)}{\Gamma \vdash x : \tau} \qquad \frac{}{\Gamma \vdash n : \texttt{int}} \cdots \qquad \frac{\tau \leq type(op)}{\Gamma \vdash op : \tau}$$

$$\frac{\Gamma + x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \texttt{fun}\ x \rightarrow e : \tau_1 \rightarrow \tau_2} \qquad \frac{\Gamma \vdash e_1 : \tau' \rightarrow \tau \quad \Gamma \vdash e_2 : \tau'}{\Gamma \vdash e_1\ e_2 : \tau}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma + x : Gen(\tau_1, \Gamma) \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau_2}$$

deux opérations apparaissent

- l'**instanciation**, dans la règle

$$\frac{\tau \leq \Gamma(x)}{\Gamma \vdash x : \tau}$$

la relation $\tau \leq \sigma$ se lit « τ est une instance de σ » et est définie par

$$\tau \leq \forall \alpha_1 ... \alpha_n . \tau' \quad \text{ssi} \quad \exists \tau_1 ... \exists \tau_n .\ \tau = \tau'[\alpha_1 \leftarrow \tau_1, ..., \alpha_n \leftarrow \tau_n]$$

exemple : $\texttt{int} \times \texttt{bool} \rightarrow \texttt{int} \leq \forall \alpha . \forall \beta . \alpha \times \beta \rightarrow \alpha$

- et la **généralisation**, dans la règle

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma + x : \mathit{Gen}(\tau_1, \Gamma) \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau_2}$$

  où

$$\mathit{Gen}(\tau_1, \Gamma) \stackrel{\text{def}}{=} \forall \alpha_1 \ldots \forall \alpha_n . \tau_1 \quad \text{où} \quad \{\alpha_1, \ldots, \alpha_n\} = \mathcal{L}(\tau_1) \backslash \mathcal{L}(\Gamma)$$

$$\dfrac{\dfrac{\dfrac{\alpha \leq \alpha}{x : \alpha \vdash x : \alpha}}{\emptyset \vdash \texttt{fun}\ x \rightarrow x : \alpha \rightarrow \alpha} \qquad \dfrac{\dfrac{\dfrac{\texttt{int}\rightarrow\texttt{int} \leq \forall\alpha.\alpha\rightarrow\alpha}{\Gamma\vdash f:\texttt{int}\rightarrow\texttt{int}} \quad \vdots}{\Gamma \vdash f\ 1 : \texttt{int}} \quad \dfrac{\dfrac{\texttt{bool}\rightarrow\texttt{bool} \leq \forall\alpha.\alpha\rightarrow\alpha}{\Gamma\vdash f:\texttt{bool}\rightarrow\texttt{bool}} \quad \vdots}{\Gamma \vdash f\ \texttt{true} : \texttt{bool}}}{\Gamma \vdash (f\ 1, f\ \texttt{true}) : \texttt{int} \times \texttt{bool}}}{\emptyset \vdash \texttt{let}\ f = \texttt{fun}\ x \rightarrow x\ \texttt{in}\ (f\ 1, f\ \texttt{true}) : \texttt{int} \times \texttt{bool}}$$

avec

$$\begin{array}{rcl} \Gamma & \stackrel{\text{def}}{=} & \emptyset + f : \mathit{Gen}(\alpha \rightarrow \alpha, \emptyset) \\ & = & f : \forall\alpha.\alpha \rightarrow \alpha \end{array}$$

pour inférer le type d'une expression, il reste des problèmes

- dans `fun` $x \rightarrow e$, quel type donner à x ?
- pour une variable x, quelle instance de $\Gamma(x)$ choisir ?

il existe une solution : **l'algorithme W** (Milner, Damas, Tofte) deux idées

- on utilise de **nouvelles variables de types** pour représenter les types inconnus
  - pour le type de x dans `fun` $x \rightarrow e$
  - pour instancier les variables du schéma $\Gamma(x)$

- on détermine la valeur de ces variables plus tard, par **unification entre types** au moment du typage de l'application soient deux types τ_1 et τ_2 contenant des variables de types $\alpha_1, \dots, \alpha_n$

existe-t-il une instanciation f des variables α_i telle que $f(\tau_1) = f(\tau_2)$?

on appelle cela l'**unification**

exemple 1 :

$$
\begin{array}{l}
\tau_1 = \alpha \times \beta \to \mathtt{int} \\
\tau_2 = \mathtt{int} \times \mathtt{bool} \to \gamma \\
\text{solution :} \quad \alpha = \mathtt{int} \ \wedge \ \beta = \mathtt{bool} \ \wedge \ \gamma = \mathtt{int}
\end{array}
$$

exemple 2 :

$$
\begin{array}{l}
\tau_1 = \alpha \times \mathtt{int} \to \alpha \times \mathtt{int} \\
\tau_2 = \gamma \to \gamma \\
\text{solution :} \quad \gamma = \alpha \times \mathtt{int}
\end{array}
$$

exemple 3 :

$$\tau_1 = \alpha \rightarrow \texttt{int}$$
$$\tau_2 = \beta \times \gamma$$

pas de solution

exemple 4 :

$$\tau_1 = \alpha \rightarrow \texttt{int}$$
$$\tau_2 = \alpha$$

pas de solution

unifier(τ_1, τ_2) détermine s'il existe une instance des variables de types de τ_1 et τ_2 telle que $\tau_1 = \tau_2$

$$
\begin{array}{rcl}
\mathit{unifier}(\tau, \tau) & = & \text{succès} \\[1em]
\mathit{unifier}(\tau_1 \to \tau_1', \tau_2 \to \tau_2') & = & \mathit{unifier}(\tau_1, \tau_2)\ ;\ \mathit{unifier}(\tau_1', \tau_2') \\[1em]
\mathit{unifier}(\tau_1 \times \tau_1', \tau_2 \times \tau_2') & = & \mathit{unifier}(\tau_1, \tau_2)\ ;\ \mathit{unifier}(\tau_1', \tau_2') \\[1em]
\mathit{unifier}(\alpha, \tau) & = & \text{si } \alpha \notin \mathcal{L}(\tau),\ \text{remplacer } \alpha \text{ par } \tau \text{ partout} \\
& & \text{sinon, échec} \\[1em]
\mathit{unifier}(\tau, \alpha) & = & \mathit{unifier}(\alpha, \tau) \\[1em]
\mathit{unifier}(\tau_1, \tau_2) & = & \text{échec dans tous les autres cas}
\end{array}
$$

(pas de panique : on le fera en TD)

l'idée de l'algorithme W sur un exemple

considérons l'expression $\boxed{\texttt{fun}\ x \rightarrow +(\mathit{fst}\ x, 1)}$

- à x on donne le type α_1, nouvelle variable de type
- la primitive $+$ a le type $\texttt{int} \times \texttt{int} \rightarrow \texttt{int}$
- typons l'expression $(\mathit{fst}\ x, 1)$
  - fst a pour type le schéma $\forall\alpha.\forall\beta.\alpha \times \beta \rightarrow \alpha$
  - on lui donne donc le type $\alpha_2 \times \beta_1 \rightarrow \alpha_2$ pour deux nouvelles variables
  - l'application $\mathit{fst}\ x$ impose d'unifier α_1 et $\alpha_2 \times \beta_1$, $\Rightarrow \alpha_1 = \alpha_2 \times \beta_1$
- $(\mathit{fst}\ x, 1)$ a donc le type $\alpha_2 \times \texttt{int}$
- l'application $+(\mathit{fst}\ x, 1)$ unifie $\texttt{int} \times \texttt{int}$ et $\alpha_2 \times \texttt{int}$,
  $\Rightarrow \alpha_2 = \texttt{int}$

au final, on obtient le type $\texttt{int} \times \beta_1 \rightarrow \texttt{int}$,
et si on généralise (dans un `let`) on obtient donc $\boxed{\forall\beta.\texttt{int} \times \beta \rightarrow \texttt{int}}$

on définit une fonction W qui prend en arguments un environnement Γ et une expression e et renvoie le type inféré pour e

- si e est une variable x,
  renvoyer une instance triviale de $\Gamma(x)$
- si e est une constante c,
  renvoyer une instance triviale de son type (penser à $[] : \alpha\ \texttt{list}$)
- si e est une primitive op,
  renvoyer une instance triviale de son type
- si e est une paire (e_1, e_2),
  calculer $\tau_1 = W(\Gamma, e_1)$
  calculer $\tau_2 = W(\Gamma, e_2)$
  renvoyer $\tau_1 \times \tau_2$

- si e est une fonction `fun` $x \rightarrow e_1$,
  soit α une nouvelle variable
  calculer $\tau = W(\Gamma + x : \alpha, e_1)$
  renvoyer $\alpha \rightarrow \tau$

- si e est une application $e_1\ e_2$,
  calculer $\tau_1 = W(\Gamma, e_1)$
  calculer $\tau_2 = W(\Gamma, e_2)$
  soit α une nouvelle variable
  unifier(τ_1, $\tau_2 \rightarrow \alpha$)
  renvoyer α

- si e est `let` $x = e_1$ `in` e_2,
  calculer $\tau_1 = W(\Gamma, e_1)$
  renvoyer $W(\Gamma + x : Gen(\tau_1, \Gamma), e_2)$

Théorème (Damas, Milner, 1982)

L'algorithme W est correct, complet et détermine le type principal :

- *si* $W(\emptyset, e) = \tau$ *alors* $\emptyset \vdash e : \tau$
- *si* $\emptyset \vdash e : \tau$ *alors* $\tau \leq \forall\bar{\alpha}.W(\emptyset, e)$

Théorème (sûreté du typage)

Le système de Hindley-Milner est sûr.
(Si $\emptyset \vdash e : \tau$*, alors la réduction de e est infinie*
ou se termine sur une valeur.)

il existe plusieurs façons de réaliser l'unification

- en manipulant explicitement une **substitution**

```
type tvar = int
type subst = typ TVmap.t
```

- en utilisant des variables de types **destructives**

```
type tvar = { id: int; mutable def: typ option; }
```

il existe également plusieurs façons de coder l'algorithme W

- avec des **schémas explicites** et en calculant $Gen(\tau, \Gamma)$

```
type schema = { tvars: TVset.t; typ: typ; }
```

- avec des **niveaux**

$$\frac{\Gamma \vdash_{n+1} e_1 : \tau_1 \qquad \Gamma + x : (\tau_1, n) \vdash_n e_2 : \tau_2}{\Gamma \vdash_n \texttt{let } x = e_1 \texttt{ in } e_2 : \tau_2}$$

cf. TD

on peut étendre mini-ML de nombreuses façons

- récursion
- types construits (*n*-uplets, listes, types sommes et produits)
- références

comme déjà expliqué, on peut définir

$$\texttt{let rec } f\ x = e_1 \texttt{ in } e_2 \stackrel{\text{def}}{=} \texttt{let } f = \mathit{opfix}\ (\texttt{fun } f \to \texttt{fun } x \to e_1) \texttt{ in } e_2$$

où

$$\mathit{opfix} : \forall\alpha.\forall\beta.((\alpha \to \beta) \to (\alpha \to \beta)) \to (\alpha \to \beta)$$

de manière équivalente, on peut donner la règle

$$\frac{\Gamma + f : \tau \to \tau_1 + x : \tau \vdash e_1 : \tau_1 \qquad \Gamma + f : \mathit{Gen}(\tau \to \tau_1, \Gamma) \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{let rec } f\ x = e_1 \texttt{ in } e_2 : \tau_2}$$

on a déjà vu les paires

les listes ne posent pas de difficulté

$$
\begin{array}{rl}
\texttt{[]}: & \forall\alpha.\,\alpha\ \texttt{list} \\
\texttt{::}: & \forall\alpha.\,\alpha \times \alpha\ \texttt{list} \rightarrow \alpha\ \texttt{list}
\end{array}
$$

$$
\frac{\Gamma \vdash e_1 : \tau\ \texttt{list} \qquad \Gamma \vdash e_2 : \tau_1 \qquad \Gamma + x : \tau + y : \tau\ \texttt{list} \vdash e_3 : \tau_1}{\Gamma \vdash \texttt{match}\ e_1\ \texttt{with}\ \texttt{[]} \rightarrow e_2 \mid \texttt{::}(x, y) \rightarrow e_3 : \tau_1}
$$

se généralise aisément aux types sommes et produits pour les références, on peut naïvement penser qu'il suffit d'ajouter les primitives

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

malheureusement, c'est incorrect !

$$\begin{array}{ll} \texttt{let } r = \texttt{ref } (\texttt{fun } x \rightarrow x) \texttt{ in} & r : \forall\alpha.(\alpha \rightarrow \alpha) \texttt{ ref} \\ \texttt{let } \_ = r \texttt{ := } (\texttt{fun } x \rightarrow x\ 1) \texttt{ in} & \\ !r \texttt{ true} & \textit{boum !} \end{array}$$

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` *dont le type est généralisé est de la forme*

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

où v_1 est une **valeur**.

en pratique, on continue d'écrire

$$\texttt{let } r = \texttt{ref } (\texttt{fun } x \rightarrow x) \texttt{ in } \ldots$$

mais le type de r n'est pas généralisé

comme si on avait écrit

$$(\texttt{fun } r \rightarrow \ldots) \; (\texttt{ref } (\texttt{fun } x \rightarrow x))$$

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 de la nature du type `'a t` : non pour un tableau, oui pour une liste, 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*
  (également en rapport avec le cours 2)
- Xavier Leroy et Pierre Weis, *Le langage Caml*
  (le dernier chapitre explique l'inférence avec les niveaux)

la suite

- TD 6
  - unification et algorithme W

- prochain cours
  - mode de passage des paramètres