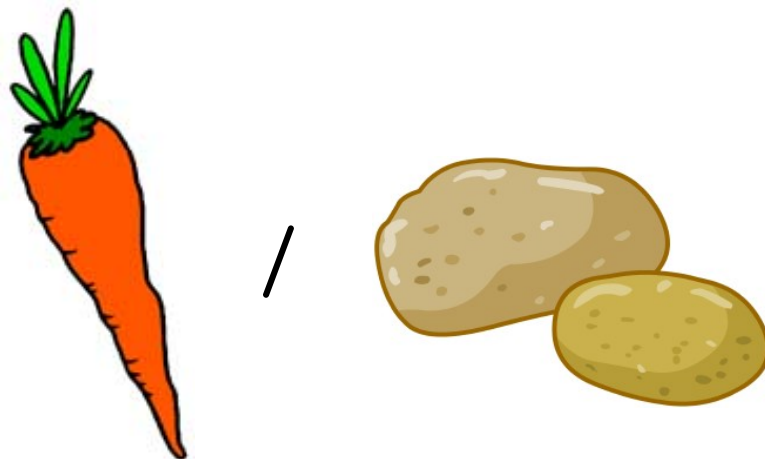
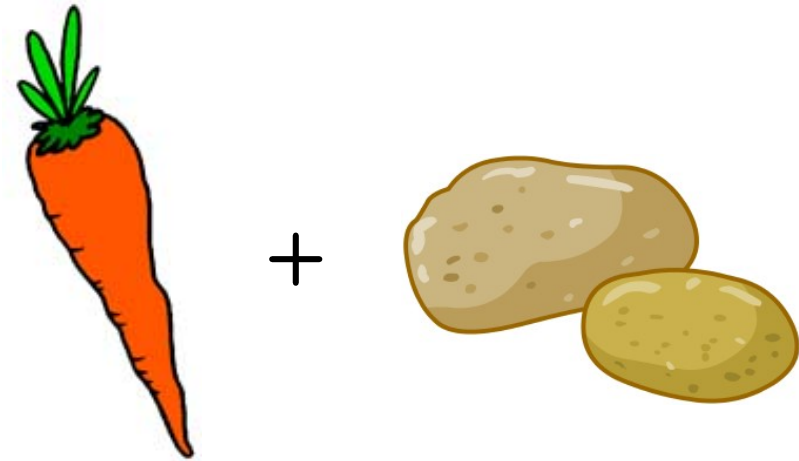


# Langages de programmation et compilation

Jean-Christophe Filliâtre

Cours 4 / 20 octobre 2011



si j'écris l'expression

"5" + 37

dois-je obtenir

- une erreur ? (Caml)
- l'entier 42 ? (Visual Basic)
- 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 statique du programme qui associe un **type** à chaque sous-expression, dans le but de rejeter les programmes incohérents mais aussi de permettre la compilation

*well typed programs do not go wrong*

## Objectifs du typage

- 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**

## Plusieurs solutions

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

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

facile de vérifier mais trop fastidieux pour le programmeur

- 2 annoter seulement les 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 ⇒ **inférence** complète (Caml, 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

## mini-ML

## Typage monomorphe de mini-ML

rappel

$e ::= x$	identificateur
$c$	constante (1, 2, ..., true, ...)
$op$	primitive (+, ×, fst, ...)
$\text{fun } x \rightarrow e$	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

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

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

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

et se lit « dans l'environnement  $\Gamma$ , l'expression  $e$  a le type  $\tau$  »

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

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

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

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

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

## Exemple

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

## Expressions non typables

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  $\text{fun } x \rightarrow x x$

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

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 = fun x → x in (f 1, f true)` n'est pas typable  
car il n'y a pas de type  $\tau$  tel que

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

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*

## Fonction récursive

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 `fun x → 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  $\Gamma$

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

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  $\Gamma$  est réalisé par une structure persistante

en l'occurrence on utilise le module Map de Caml

```
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".

## Sûreté du typage

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  $e \xrightarrow{*} e'$  et  $e'$  irréductible, alors  $e'$  est une valeur.*

## Sûreté du typage

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

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

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  $e'$  ( $\alpha$ -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  $\text{fun } x \rightarrow x$  dans

$\text{let } f = \text{fun } x \rightarrow x \text{ in } \dots$

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

une solution : le **polymorphisme paramétrique**

## Polymorphisme paramétrique

## Système F

on étend l'algèbre des types :

$\tau ::=$	<code>int   bool   ...</code>	<i>types de base</i>
	$\tau \rightarrow \tau$	<i>type d'une fonction</i>
	$\tau \times \tau$	<i>type d'une paire</i>
	$\alpha$	<i>variable de type</i>
	$\forall \alpha. \tau$	<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  $\Gamma$ , défini par

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

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

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

## Primitives

## Exercice

on peut maintenant donner un type satisfaisant aux primitives

$$\text{fst} : \forall \alpha. \forall \beta. \alpha \times \beta \rightarrow \alpha$$

$$\text{snd} : \forall \alpha. \forall \beta. \alpha \times \beta \rightarrow \beta$$

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

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

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

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

et on accepterait donc le programme

`(fun x → x) 1 2`

problème : l'inférence de type dans le système F n'est pas décidable

même si  $e$  et  $\tau$  sont donnés, il n'est pas possible de vérifier si  $\vdash e : \tau$  (pour un terme  $e$  sans annotations)

pour obtenir une inférence de types décidable, on va restreindre la puissance du système F  $\Rightarrow$  le système de **Hindley-Milner**, utilisé dans Caml, SML, Haskell, ...

## Le système de Hindley-Milner

## Exemples

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

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

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

dans Hindley-Milner, les types suivants sont toujours acceptés

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

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 préfixe est implicite

```
# fst;;
```

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

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

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

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

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

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

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

## Le système de Hindley-Milner

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 = fun x -> x in (f 1, f true)
```

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

## Le système de Hindley-Milner

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 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

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

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 retour en arrière ?

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

$$\frac{\tau \leq \Gamma(x)}{\Gamma \vdash x : \tau} \quad \frac{}{\Gamma \vdash n : \text{int}} \quad \frac{\tau \leq \text{type}(op)}{\Gamma \vdash op : \tau}$$

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

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

deux opérations apparaissent

- l'**instanciation**, dans la règle

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

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

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

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

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

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

où

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

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

avec  $\Gamma' = \Gamma + f : Gen(\alpha \rightarrow \alpha, \Gamma) = \Gamma + f : \forall \alpha. \alpha \rightarrow \alpha$   
si  $\alpha$  est choisie non libre dans  $\Gamma$

## Inférence de types pour mini-ML

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

- dans  $\text{fun } x \rightarrow e$ , 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)

## L'algorithme W

deux idées

- on utilise de **nouvelles variables de types** pour représenter les types inconnus
  - pour le type de  $x$  dans  $\text{fun } x \rightarrow e$
  - 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  $\tau_1$  et  $\tau_2$  contenant des variables de types  $\alpha_1, \dots, \alpha_n$

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

on appelle cela l'**unification**

exemple 1 :

$\tau_1 = \alpha \times \beta \rightarrow \text{int}$   
 $\tau_2 = \text{int} \times \text{bool} \rightarrow \gamma$   
 solution :  $\alpha = \text{int} \wedge \beta = \text{bool} \wedge \gamma = \text{int}$

exemple 2 :

$\tau_1 = \alpha \times \text{int} \rightarrow \alpha \times \text{int}$   
 $\tau_2 = \gamma \rightarrow \gamma$   
 solution :  $\gamma = \alpha \times \text{int}$

exemple 3 :

$\tau_1 = \alpha \rightarrow \text{int}$   
 $\tau_2 = \beta \times \gamma$   
 pas de solution

exemple 4 :

$\tau_1 = \alpha \rightarrow \text{int}$   
 $\tau_2 = \alpha$   
 pas de solution

## Pseudo-code de l'unification

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

$\text{unifier}(\tau, \tau) = \text{succès}$

$\text{unifier}(\tau_1 \rightarrow \tau'_1, \tau_2 \rightarrow \tau'_2) = \text{unifier}(\tau_1, \tau_2) ; \text{unifier}(\tau'_1, \tau'_2)$

$\text{unifier}(\tau_1 \times \tau'_1, \tau_2 \times \tau'_2) = \text{unifier}(\tau_1, \tau_2) ; \text{unifier}(\tau'_1, \tau'_2)$

$\text{unifier}(\alpha, \tau) =$  si  $\alpha \notin \mathcal{L}(\tau)$ , remplacer  $\alpha$  par  $\tau$  partout  
 sinon, échec

$\text{unifier}(\tau, \alpha) = \text{unifier}(\alpha, \tau)$

$\text{unifier}(\tau_1, \tau_2) =$  échec dans tous les autres cas

(pas de panique : on le fera en TD)

## L'idée de l'algorithme W sur un exemple

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

- à  $x$  on donne le type  $\alpha_1$ , nouvelle variable de type
- la primitive  $+$  a le type  $\text{int} \times \text{int} \rightarrow \text{int}$
- typons l'expression  $(fst\ x, 1)$ 
  - $fst$  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  $fst\ x$  impose d'unifier  $\alpha_1$  et  $\alpha_2 \times \beta_1$ ,  $\Rightarrow \alpha_1 = \alpha_2 \times \beta_1$
- $(fst\ x, 1)$  a donc le type  $\alpha_2 \times \text{int}$
- l'application  $+(fst\ x, 1)$  unifie  $\text{int} \times \text{int}$  et  $\alpha_2 \times \text{int}$ ,  $\Rightarrow \alpha_2 = \text{int}$

au final, on obtient le type  $\text{int} \times \beta_1 \rightarrow \text{int}$ ,

et en généralisant on obtient finalement  $\boxed{\forall \beta. \text{int} \times \beta \rightarrow \text{int}}$

on définit une fonction  $W$  qui prend en argument un environnement  $\Gamma$  et une expression  $e$ , et retourne 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 `[] :  $\alpha$  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  $\alpha$  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  $\alpha$  une nouvelle variable  
unifier( $\tau_1, \tau_2 \rightarrow \alpha$ )  
renvoyer  $\alpha$
- si  $e$  est `let  $x = e_1$  in  $e_2$` ,  
calculer  $\tau_1 = W(\Gamma, e_1)$   
renvoyer  $W(\Gamma + x : Gen(\tau_1, \Gamma), e_2)$

## Résultats

### 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  $\tau \leq \forall \alpha. W(\emptyset, e)$

### 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.)*

## Considérations algorithmiques

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**

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

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

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

où

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

de manière équivalente, on peut donner la règle

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

## Types construits

on a déjà vu les paires

les listes ne posent pas de difficulté

$$\begin{aligned} \text{nil} &: \forall \alpha. \alpha \text{ list} \\ \text{cons} &: \forall \alpha. \alpha \times \alpha \text{ list} \rightarrow \alpha \text{ list} \end{aligned}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \text{ list} \quad \Gamma \vdash e_2 : \tau \quad \Gamma + x : \tau_1 + y : \tau_1 \text{ list} \vdash e_3 : \tau}{\Gamma \vdash \text{match } e_1 \text{ with } \text{nil} \rightarrow e_2 \mid \text{cons}(x, y) \rightarrow e_3 : \tau}$$

se généralise aisément aux types sommes et produits

## Références

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 !

$$\begin{aligned} \text{let } r &= \text{ref} \ (\text{fun } x \rightarrow x) \text{ in} & r &: \forall \alpha. (\alpha \rightarrow \alpha) \text{ ref} \\ \text{let } \_ &= r := \ (\text{fun } x \rightarrow x + 1) \text{ in} & & \\ !r &\text{ true} & \text{bom !} & \end{aligned}$$

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

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

mais on peut écrire en revanche

$$(\text{fun } r \rightarrow \dots) (\text{ref } (\text{fun } x \rightarrow x))$$

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 Caml, 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 : expander pour faire apparaître une fonction, *i.e.*, une valeur

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

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

(on parle d' $\eta$ -expansion)

en présence du système de modules, la réalité est plus complexe encore étant donné un module M

```
module M : sig
  type  $\alpha$  t
  val create : int ->  $\alpha$  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 Caml, une indication de **variance** permet de distinguer les deux

```
type + $\alpha$  t (* on peut généraliser *)
type  $\alpha$  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 3)
- Xavier Leroy et Pierre Weis, *Le langage Caml* (le dernier chapitre explique l'inférence avec les niveaux)

- la semaine prochaine
  - TD de mercredi : unification et algorithme W
  - **pas de cours** le 27 octobre
- la semaine suivante
  - Cours 5 le jeudi 3 novembre : Analyse lexicale

ce cours reprend des notes de cours de Xavier Leroy  
(cours de remise à niveau « Programmation, sémantique, typage »)