

---

# INFÉRENCE DE TYPE

## *Épreuve orale d'informatique du CCMP*

---

On s'intéresse dans ce sujet au problème de déterminer automatiquement le type d'une expression, ou de détecter si elle n'est pas typable, dans un langage typé à la OCaml. Tous les programmes sont à écrire en OCaml, dans le fichier `inference.ml` fourni. Ce fichier contient un certain nombre de tests, qu'il ne faut pas hésiter à compléter avec les vôtres.

### 1 Dédution naturelle

On rappelle les règles de la déduction naturelle :

#### Règles structurelles

$$\frac{}{\Gamma, \varphi \vdash \varphi} (\text{Ax}) \qquad \frac{\Gamma \vdash \varphi}{\Gamma, \psi \vdash \varphi} (\text{Aff})$$

#### Règles d'introduction

$$\frac{}{\Gamma \vdash \top} (\top_i) \qquad \frac{\Gamma, \varphi \vdash \psi}{\Gamma \vdash \varphi \rightarrow \psi} (\rightarrow_i)$$

$$\frac{\Gamma \vdash \varphi}{\Gamma \vdash \varphi \vee \psi} (\vee_i^g) \qquad \frac{\Gamma \vdash \psi}{\Gamma \vdash \varphi \vee \psi} (\vee_i^d)$$

$$\frac{\Gamma \vdash \varphi \quad \Gamma \vdash \psi}{\Gamma \vdash \varphi \wedge \psi} (\wedge_i) \qquad \frac{\Gamma, \varphi \vdash \perp}{\Gamma \vdash \neg \varphi} (\neg_i)$$

#### Règles d'élimination

$$\frac{\Gamma \vdash \varphi \rightarrow \psi \quad \Gamma \vdash \varphi}{\Gamma \vdash \psi} (\rightarrow_e) \qquad \frac{\Gamma \vdash \varphi \vee \psi \quad \Gamma, \varphi \vdash \theta \quad \Gamma, \psi \vdash \theta}{\Gamma \vdash \theta} (\vee_e)$$

$$\frac{\Gamma \vdash \varphi \wedge \psi}{\Gamma \vdash \varphi} (\wedge_e^g) \qquad \frac{\Gamma \vdash \varphi \wedge \psi}{\Gamma \vdash \psi} (\wedge_e^d)$$

$$\frac{\Gamma \vdash \neg \varphi \quad \Gamma \vdash \varphi}{\Gamma \vdash \perp} (\neg_e) \qquad \frac{\Gamma \vdash \perp}{\Gamma \vdash \varphi} (\perp_e)$$

## Règles de la logique classique

$$\frac{\Gamma, \neg\varphi \vdash \perp}{\Gamma \vdash \varphi} \text{ (Abs)} \quad \frac{}{\Gamma \vdash \varphi \vee \neg\varphi} \text{ (TE)} \quad \frac{\Gamma \vdash \neg\neg\varphi}{\Gamma \vdash \varphi} (\neg\neg_e)$$

**Question 1** Donner une dérivation en déduction naturelle du séquent

$$\vdash ((a \rightarrow b) \wedge (b \rightarrow c)) \rightarrow (a \vee b) \rightarrow c$$

## 2 Structure persistante de dictionnaire

Nous aurons besoin dans la suite de dictionnaires pour conserver en mémoire le type d'une variable, ou représenter une substitution.

**Question 2** Implémenter une telle structure de dictionnaire à l'aide d'arbres binaires de recherche dans un type ('a, 'b) dict, 'a étant le type des clés et 'b celui des valeurs. Seules les opérations de création d'un dictionnaire vide, de récupération de la valeur associée à une clé, et d'insertion d'une clé et d'une valeur, seront nécessaires.

À tout moment du sujet, on pourra décider de revenir sur cette implémentation pour utiliser des arbres bicolores à la place.

On précisera la complexité de chaque opération.

## 3 Inférence avec annotation des paramètres

On va représenter le langage à typer par des objets du type `expression1`, pour lesquels on souhaite déterminer un objet du type `typ1` :

```
type typ1 =
| Int (* type de base des entiers *)
| Arrow of typ1 * typ1 (* t1 -> t2 *)
| Product of typ1 * typ1 (* t1 * t2 *);;

type expression1 =
| Var of string
| Const of int
| Op of string (* opérateur prenant un couple d'entiers
                et renvoyant un entier *)
| Fun of string * typ1 * expression1 (* fun (x : t) -> e *)
| App of expression1 * expression1 (* e1 e2 *)
| Couple of expression1 * expression1 (* e1, e2 *)
| Let of string * expression1 * expression1 (* let x = e1 in e2 *);;
```

On note que dans cette première version, afin de simplifier le problème, le type de l'argument d'une fonction doit apparaître explicitement. Toujours pour simplifier, on ne considère qu'un seul type de base, `Int`, correspondant au type des entiers `int`. On ne considérera que des opérations correspondant au type `(int * int) -> int` (par exemple `Op "+"`).

**Question 3** Avec un système de types à la OCaml, l'expression suivante est-elle typable, et si oui de quel type ?

```
Let ("f",
Fun ("x", Int, App (Op "+", Couple (Var "x", Const 1))),
App (Var "f", Const 2))
```

On donne les règles suivantes, permettant de dériver des jugements de typage sur le même modèle qu'une preuve en déduction naturelle ( $x$  désigne une variable,  $n$  une constante,  $+$  une opération) :

$$\begin{array}{c}
\frac{}{\Gamma, x : t \vdash x : t} \qquad \frac{}{\Gamma \vdash n : \text{int}} \qquad \frac{}{\Gamma \vdash + : (\text{int} * \text{int}) \rightarrow \text{int}} \\
\\
\frac{\Gamma, x : t_1 \vdash e : t_2}{\Gamma \vdash (\text{fun } (x : t_1) \rightarrow e) : t_1 \rightarrow t_2} \qquad \frac{\Gamma \vdash e_1 : t' \rightarrow t \quad \Gamma \vdash e_2 : t'}{\Gamma \vdash e_1 e_2 : t} \\
\\
\frac{\Gamma \vdash e_1 : t_1 \quad \Gamma \vdash e_2 : t_2}{\Gamma \vdash (e_1, e_2) : (t_1 * t_2)} \qquad \frac{\Gamma \vdash e_1 : t_1 \quad \Gamma, x : t_1 \vdash e_2 : t_2}{\Gamma \vdash (\text{let } x = e_1 \text{ in } e_2) : t_2}
\end{array}$$

**Question 4** Dériver le typage de l'expression précédente dans ce système.

**Question 5** Écrire une fonction `print_typ1 : typ1 -> unit` et une fonction `print_expression1 : expression1 -> unit` affichant un type et une expression. L'expression précédente pourrait ainsi être affichée comme :

```
let f = fun x : int -> + (x, 1) in f 2.
```

**Question 6** Écrire une fonction `typ_expr : expression1 -> typ1` prenant en argument une expression et renvoyant son type. La fonction lèvera une erreur si l'expression n'est pas typable.

**Indication :** La fonction sera récursive et suivra les règles du système précédent. L'environnement  $\Gamma$  sera encodé par un `(string, typ1) dict`.

## 4 Inférence sans annotation

On considère maintenant un système de typage dans lequel les types des arguments ne sont plus annotés, et où les types peuvent contenir des variables de type, par exemple `fun x -> x` de type `'a -> 'a`. On représente ces variables de type par des entiers, et on définit donc les types OCaml suivant :

```
type typ2 =
| Int
| Arrow of typ2 * typ2
| Product of typ2 * typ2
| Tvar of int;; (* nouveau constructeur : variable de type *)

type expression2 =
| Var of string
```

```

| Const of int
| Op of string
| Fun of string * expression2 (* plus d'annotation de type *)
| App of expression2 * expression2
| Couple of expression2 * expression2
| Let of string * expression2 * expression2;;

```

**Question 7** Écrire les fonctions `print_typ2` et `print_expression2` correspondantes.

Dans toute la suite de cette partie, on ne considère que des expressions ne contenant pas de `Let`. Les règles du système de typage sont alors :

$$\begin{array}{c}
\overline{\Gamma, x : t \vdash x : t} \qquad \overline{\Gamma \vdash n : \text{int}} \qquad \overline{\Gamma \vdash + : (\text{int} * \text{int}) \rightarrow \text{int}} \\
\\
\frac{\Gamma, x : t_1 \vdash e : t_2}{\Gamma \vdash (\text{fun } x \rightarrow e) : t_1 \rightarrow t_2} \qquad \frac{\Gamma \vdash e_1 : t' \rightarrow t \quad \Gamma \vdash e_2 : t'}{\Gamma \vdash e_1 \ e_2 : t} \qquad \frac{\Gamma \vdash e_1 : t_1 \quad \Gamma \vdash e_2 : t_2}{\Gamma \vdash (e_1, e_2) : (t_1 * t_2)}
\end{array}$$

**Question 8** Donner une dérivation du jugement de type

$$\vdash \text{fun } f \rightarrow \text{fun } g \rightarrow \text{fun } x \rightarrow f(g\ x) : (1 \rightarrow 2) \rightarrow (0 \rightarrow 1) \rightarrow 0 \rightarrow 2$$

où 0, 1 et 2 sont des variables de type.

**Remarque :** cette expression peut recevoir d'autres types, par exemple

$$(\text{int} \rightarrow \text{int}) \rightarrow (\text{int} \rightarrow \text{int}) \rightarrow \text{int} \rightarrow \text{int}$$

Tous ces autres types possibles peuvent se voir comme des cas particuliers du premier type, au sens où ils s'obtiennent par une substitution des variables de type. Ce premier type est donc un type le plus général pour cette expression, et c'est un tel type le plus général que l'on cherche à déterminer.

Puisque les types des arguments d'une fonction ne sont plus annotés, nous allons devoir initialement associer une variable de type fraîche (i.e., qui n'a pas encore été utilisée) à une variable argument. Cependant, une contrainte sur le type de cette variable argument peut apparaître dans la suite de la dérivation. Il faut alors stocker en mémoire une substitution, i.e., un dictionnaire associant à un numéro de variable de type, un type le plus général dicté par ces contraintes.

**Question 9** Écrire une fonction `appliquer : (int, typ2) dict -> typ2 -> typ2` prenant en argument une telle substitution et un type, et remplaçant dans ce type chaque variable de type apparaissant dans la substitution par son type associé.

Pour capturer les contraintes de type, nous allons utiliser l'algorithme d'unification de deux types : il s'agit de trouver une substitution la plus générale qui rende les deux types égaux, ou de déterminer que les deux types ne sont pas unifiables.

Par exemple, les types  $0 \rightarrow \text{int}$  et  $1 \rightarrow 2$  sont unifiés par la substitution  $0 \mapsto 1; 2 \mapsto \text{int}$ .

Les types  $0 \rightarrow \text{int}$  et  $1 * 2$  ne sont pas unifiables.

**Question 10** Écrire une fonction récursive

`unification : (int, typ2) dict -> typ2 -> typ2 -> (int, typ2) dict`

prenant en argument la substitution codant les contraintes déjà existantes, les deux types, et renvoyant la nouvelle substitution, ou levant une erreur s'ils ne sont pas unifiables.

On pourra suivre les indications suivantes ( $t$  désigne un type,  $\alpha$  une variable de type) :

- sur  $t, t$ , renvoyer la substitution courante ;
- sur  $t_1 \rightarrow t'_1, t_2 \rightarrow t'_2$ , unifier  $t_1$  et  $t_2$ , puis  $t'_1$  et  $t'_2$  ;
- sur  $t_1 * t'_1, t_2 * t'_2$ , idem ;
- sur  $\alpha, t$ , ou  $t, \alpha$  : si  $\alpha$  est une variable libre de  $t$ , échouer, sinon, substituer  $\alpha$  par  $t$  partout où  $\alpha$  apparaît dans la substitution et lui ajouter  $\alpha \mapsto t$  ;
- dans tous les autres cas, échouer.

L'algorithme W détermine un type le plus général d'une expression sur un principe similaire à la partie précédente, à la différence qu'une variable de type fraîche est affectée à la variable argument d'une fonction, et qu'on met progressivement à jour une substitution rendant compte des contraintes sur ces variables de type.

**Question 11** Écrire une fonction `w : expression2 -> typ2` implémentant cet algorithme. On rappelle qu'on ne considère que des expressions sans `Let`.

## 5 Inférence avec polymorphisme

Lors du typage d'une construction `Let`, il faut pouvoir généraliser le type attribué à la variable introduite, pour que chaque occurrence de cette variable puisse recevoir une instantiation indépendante de ce type. On a par exemple le jugement de type suivant :

$$\text{let } f = \text{fun } x \rightarrow x \text{ in } (f \ 1, f \ (1, 1)) : \text{int} * (\text{int} * \text{int})$$

On se donne alors le système de typage suivant, avec une nouvelle règle pour le `Let` et une règle modifiée pour les variables :

$$\frac{}{\Gamma \vdash + : (\text{int} * \text{int}) \rightarrow \text{int}}$$

$$\frac{}{\Gamma \vdash n : \text{int}}$$

$$\frac{\Gamma, x : t_1 \vdash e : t_2}{\Gamma \vdash (\text{fun } x \rightarrow e) : t_1 \rightarrow t_2}$$

$$\frac{\Gamma \vdash e_1 : t' \rightarrow t \quad \Gamma \vdash e_2 : t'}{\Gamma \vdash e_1 \ e_2 : t}$$

$$\frac{\Gamma \vdash e_1 : t_1 \quad \Gamma \vdash e_2 : t_2}{\Gamma \vdash (e_1, e_2) : (t_1 * t_2)}$$

$$\frac{\Gamma \vdash e_1 : t_1 \quad \Gamma, x : \text{Gen}(t_1, \Gamma) \vdash e_2 : t_2}{\Gamma \vdash (\text{let } x = e_1 \text{ in } e_2) : t_2}$$

$$\frac{}{\Gamma, x : (\forall a_1 \dots a_n, t) \vdash x : t[a_1 \leftarrow t_1, \dots, a_n \leftarrow t_n]}$$

avec les  $t_1, \dots, t_n$  des types quelconques, et  $\text{Gen}(t, \Gamma) = \forall a_1 \dots a_n, t$ , où  $a_1, \dots, a_n$  sont les variables libres de  $t$  qui ne sont pas des variables (de type) libres de  $\Gamma$ .

**Question 12** Donner une dérivation du jugement de type précédent dans ce système.

**Question 13** Écrire une nouvelle version de la fonction  $w$  implémentant l'algorithme  $W$  basé sur ce système.