Institut National
Universitaire
**Champollion**

**Types de données, preuves**

**Cours-TD n° 5**

# Chapitre 5 - Inférence de types

L3 Info

Laura Brillon - laura.brillon[at]univ-jfc.fr

# Chapitre 5 - Inférence de types

## Sommaire

# Mise en place

## Système de types

Nous allons retrouver les expressions du langage de programmation du chapitre sur la vérification de type :

- Constantes : `2`, `true`, `...`
- Variables : `x`, `y` , `...`
- Fonctions : `fun x : T -> e`
  où `x` est une variable de type `T` et `e` est une expression
- Applications : `(e e')`
  où `e`, `e'` sont des expressions.

*Remarque : Le langages est moins riche que lors de la vérification de types.*

# Mise en place

## Unification sur des types

On s'autorisera un certain "relâchement" dans l'application de l'algorithme d'unification sur des types.

**Exemple**

1. Unif(`'a -> bool, int -> 'b`)
   Unif(`'a, int`) = [`'a` $\leftarrow$ `int`]
   Unif(`bool, 'b`) = [`'b` $\leftarrow$ `bool`]
   = [`'a` $\leftarrow$ `int , 'b` $\leftarrow$ `bool`]
2. Unif(`int ->(int -> bool), (int*int) -> bool`)
   Unif(`int, (int * int)`) = fail
   = fail

# Inférence de types

## Sommaire

# Inférence de types

## Motivations

**Fonction annotée :**

```
fun (f : int -> int) -> f (f 3) ;;
- : (int -> int) -> int = <fun>
```

En Caml, il suffit d'écrire :

```
fun f -> f (f 3) ;;
- : (int -> int) -> int = <fun>
```

Inférence, avantages et inconvénients ?

# Inférence de types

## Observations

Est-ce que l'inférence reconstruit toujours l'annotation de type ?

**Fonction annotée :**

```
fun (f: int -> int) -> fun (x: int) -> f (f x) ;;
- : (int -> int) -> int -> int = <fun>
```

**Fonction non annotée :**

```
fun f -> fun x -> f (f x) ;;
```

$\rightsquigarrow$ Quel est le type de cette fonction ?

## Observations

Est-ce que l'inférence reconstruit toujours l'annotation de type ?

**Fonction annotée :**

```
fun (f: int -> int) -> fun (x: int) -> f (f x) ;;
- : (int -> int) -> int -> int = <fun>
```

**Fonction non annotée :**

```
fun f -> fun x -> f (f x) ;;
- : ('a -> 'a) -> 'a -> 'a = <fun>
```

**Conclusion :**

Le type inféré peut être *plus général* que le type annoté.

**Définition**

Un type `T` est dit *plus général* que $T'$ s'il existe une substitution $\sigma$ telle que $T' = \sigma(T)$.

# Inférence de types

## Cahier des charges

On se donne :

- une expression e
- un environnement de typage `Env`

L'algorithme d'inférence de types doit :

- S'il existe, calculer le type le plus général `T` tel que

$$\texttt{Env} \vdash \texttt{e} : \texttt{T}$$

- Sinon, indiquer que e n'est pas typable dans `Env`.

**Définition**

Le type le plus général d'une expression e fixée, est appelé le *type principal* de cette expression.

# Où en sommes-nous ?

## Sommaire

# Algorithme

## Idée de l'algorithme

- `fun f -> (f 42), dans Env = [ ]`
  - ↪ $'x_f$ variable de type - `EnvT = [(f, 'x_f)]`
- `(f 42), dans EnvT`
  - ↪ $'x_f$ et `int`
    - ☾ Création d'une nouvelle variable de type `'b`
    - ☾ `(f 42)` est de type `'b` avec contrainte $'x_f \stackrel{?}{=} \texttt{int} \to \texttt{'b}$
- `fun f -> (f 42)` est de type $'x_f \to$ `'b`
  avec la contrainte $'x_f \stackrel{?}{=}$ `int` $\to$ `'b` $\rightsquigarrow$ Unification
- `fun f -> (f 42)` est de type `(int` $\to$ `'b)` $\to$ `'b`

# Algorithme

## Idée de l'algorithme

### Étape 1 - Générer des contraintes

- `fun f -> (f 42), dans Env = [ ]`
  $\hookrightarrow$ '$x_f$ variable de type - `EnvT = [(f, 'x_f)]`

- `(f 42), dans EnvT`
  $\hookrightarrow$ '$x_f$ et `int`
  - Création d'une nouvelle variable de type `'b`
  - `(f 42)` est de type `'b` avec contrainte '$x_f \overset{?}{=}$ `int`$\rightarrow$`'b`

- `fun f -> (f 42)` est de type `'`$x_f \rightarrow$ `'b`
  avec la contrainte '$x_f \overset{?}{=}$ `int` $\rightarrow$ `'b` $\rightsquigarrow$ Unification

### Étape 2 - Unification des contraintes

### Étape 3 - Bilan

- `fun f -> (f 42)` est de type `(int` $\rightarrow$ `'b)` $\rightarrow$ `'b`

# Algorithme

## Cette fois, c'est la bonne !

### Étape 1 : Générer des contraintes

**Entrées :** Environnement `Env`, Expression e
**Sorties :** Type provisoire, ensemble de contraintes $\Sigma$

**Algorithme GC (GénèreContraintes) :**

- Si e=c est une constante, renvoyer (`Type(c)`, [ ]).
- Si e=v est une variable,
  - OU `(v, Type(v))` $\in$ `Env`, renvoyer (`Type(v)`, [ ])
  - OU `(v, Type(v))` $\notin$ `Env`, Erreur "Variable non définie ".
- Si `e = (f a)` est une application,
  1. Soit `(tf,cf) = GC(Env,f)`
  2. Soit `(ta,ca) = GC(Env,a)`
  3. Créer une nouvelle variable de type `'b`
  4. renvoyer(`'b`, `(tf, ta` $\rightarrow$ `'b)` $\cup$ `cf` $\cup$ `ca` )

# Algorithme

## Cette fois, c'est la bonne !

### Étape 1 : Suite

**Algorithme GC (suite) :**

- Si `e = fun v -> w` est une fonction,
  1. Créer une nouvelle variable de type `'a`
  2. Soit `(tw, cw) = GC((v,'a)::Env, w)`
  3. renvoyer (`'a` $\rightarrow$ `tw`, cw)

### Étape 2 : Unification des contraintes $\Sigma$

On détermine $\sigma = \mathit{Unif}(\Sigma)$ par l'algorithme d'unification.

### Étape 3 : Typage

Soit `(tp, Σ) = GC(Env, e)` (Étape 1)
Soit $\sigma = \mathit{Unif}(\Sigma)$ (Étape 2)
Renvoyer $tp[\sigma]$.

# Algorithme

## Exemple

### Étape 1 - Générer des contraintes

`GC (fun f -> (f 42), Env = [ ] )`

**Fonction :**

1. Nouvelle variable $'x_f$ : `EnvT = [(f, 'x_f)]`
2. `GC( (f 42), EnvT )`
   **Application :**
   - 2.1 `GC(f, EnvT) = ('x_f, [ ])`
   - 2.2 `GC(42, EnvT) = (int, [ ])`
   - 2.3 Nouvelle variable `'b`
   - 2.4 $\Rightarrow$ `('b , ['x_f` $\stackrel{?}{=}$ `int` $\rightarrow$ `'b])`
3. $\Rightarrow$ `('x_f` $\rightarrow$ `'b, ['x_f` $\stackrel{?}{=}$ `int` $\rightarrow$ `'b])`

### Étape 2 - Unification

`Unif('x_f` $\stackrel{?}{=}$ `int` $\rightarrow$ `'b ;` $\emptyset$`) = ['x_f` $\leftarrow$ `(int` $\rightarrow$ `'b) ]`

### Étape 3 - `(int` $\rightarrow$ `'b)` $\rightarrow$ `'b`