Institut National
Universitaire
Champollion

Types de données, preuves

L3 Info

Chapitre 1 - Vérification de types

Laura Brillon
laura.brillon@univ-jfc.fr

Vérification de types

Introduction

En Caml, on peut écrire

```
let f = fun x -> x + 2 ;;
f : int -> int = <fun>
```

Ici, Caml *"devine"* le type de votre fonction, c'est ce qu'on appelle

l'inférence de types.

L'algorithme d'inférence de types sera vu dans la deuxième partie du semestre.

Vérification de types

Introduction (2)

En Caml, on peut aussi écrire

```
let f = fun (x : int) -> x + 2 ;;
f : int -> int = <fun>
```

conseillé à partir de maintenant !

Ici, Caml *vérifie* que les annotations de types que vous avez écrites sont cohérentes avec le reste de la fonction.

La **vérification de type**, en programmation fonctionnelle est l'objet de ce cours.

Vérification de types

Système de types

- Définition
- Langage de types
- Environnement
- Règles de typage

Extensions

- Les couples
- Définition locale
- Type somme
- Polymorphisme

Système de types

Définition

Un système de types = deux composantes :

- Un langage de types - Quels sont les types dont on dispose ?
- Des règles de typage - Elles expriment quand une expression est bien typée.

Extrait du cours de T. Montaut :

La syntaxe générale de la sélection est

if condition then exp1 else exp2

Si la condition est de type booléen et que les deux expressions sont de même type t alors la sélection est de type t.

⇒ L'objectif est de formaliser ces règles.

Système de types

Définition

Nous allons définir notre système de types qui sera un Caml "allégé" (que l'on va enrichir dans la deuxième partie du chapitre).

Cette définition va passer par 4 étapes :

- Définition du langage de types
- Définition des règles de typage
  - Quelles expressions ?
  - Comment les typer ? (Environnement de typage)
  - Définition des règles.

Système de types

Où en est-on ?

Définition du langage de types :

Quels sont les types et les expressions que l'on s'autorise ?

Système de types

Langage de types

Quels sont les types dont on dispose ?

Types de base : `bool`, `int` , ...

Types fonctionnels : de la forme `T` $\rightarrow$ `T'`
où `T` et `T'` sont des types.

☼☼ Rappel Important : ☼☼

Convention : $\rightarrow$ associe à droite. C'est à dire,
`int` $\rightarrow$ `int` $\rightarrow$ `int` est équivalent à `int` $\rightarrow$ `(int` $\rightarrow$ `int)`.

Et donc `int` $\rightarrow$ `int` $\rightarrow$ `int` $\neq$ `(int` $\rightarrow$ `int)` $\rightarrow$ `int`.

Système de types

Question !

Question : Quel est le type des fonctions suivantes ?

```
let fA = fun x y -> x + y + 2 ;;

let fB = fun x -> (x 2) + 1 ;;
```

1. `fA : int → int → int` et `fB : int → int → int`
2. `fA : (int → int) → int` et `fB : int → int → int`
3. `fA : int → int → int` et `fB : (int → int ) → int`
4. `fA : (int → int) → int` et `fB : (int → int) → int`

Système de types

Expressions du langage de programmation

Nous avons choisi les types de notre système, maintenant, choisissons les expressions que nous pouvons écrire.

Quelles sont les expressions du langage ?

- Constantes : `2`, `true`, `...`
- Variables : `x`, `f`, `...`
- 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

Système de type

Expressions du langage de programmation

Notre langage n'est pas aussi limité qu'il n'y paraît. En effet, nous pouvons réduire à ce format :

- Les fonctions de plusieurs paramètres
  `fun (a : int) (b : int) -> a + b ;;`
  se réécrit
  `fun (a:int) -> fun (b:int) -> a + b ;;`
- Opérateurs infixes - écrire en préfixe !
  `2 + 3 ;;` se réécrit `((plus 2) 3);;`
- Les applications à plusieurs arguments
  `(plus 2 3) ;;` se réécrit `((plus 2) 3) ;;`

☼☼ Rappel Important : ☼☼

L'application associe à gauche : `f a b` $\Longleftrightarrow$ `((f a) b)`.

Système de types

Expressions du langage de programmation

Question : Quelle expression peut-on écrire dans notre système ?

1. `if (f 2) then 0 else 1 ;;`
2. `(2, ‘a‘);;`
3. `fun (x : int) -> 3 ;;`
4. `let x = 3 in x + 4 ;;`

Système de type

Où en est-on ?

Attention : Notion importante !

Système de types

Environnement de typage

Pour typer une expression, il est nécessaire de connaître le type des variables de cette expression.

$\Rightarrow$ **Environnement de typage**

Un environnement de typage associe un type à une variable.
On peut le voir comme une liste de couple (variable,type) :

$$[(v_1, T_1); \ldots; (v_n, T_n)]$$

Remarque : Ce n'est pas le même environnement qu'au S5 !

Système de types

Environnement de typage

L'environnement de typage est construit / modifié :

- lors d'une définition :

```
let f = fun (x : int) -> x + 2 ;;
f : int -> int = <fun>

let a = 3 ;;
a : int = 3
```

Environnement de typage : `[(a,int);(f, int->int)]`

- pendant la vérification de types *(... à voir)*

Système de types

Question !

Question : Quel est l'environnement de typage à l'issue de ces définitions ?

```
let f1 = fun (x : bool) -> x < true ;;
♯f1 : bool -> bool = <fun>

let x = false in f1 x ;;
♯- : bool = true

let y = f1 true ;;
♯y : bool = false
```

1. [(f, bool → bool) ; (y,bool)]
2. [(y, bool) ; (x, bool) ; (f, bool → bool)]
3. [(y, bool) ; (f, bool→ bool)]
4. [(y, false) ; (x, true) ; (f, bool → bool)]

Système de types

Où en est-on ?

Objectif : Formaliser les règles de typage de notre système.

Vérification de types

Règles de typage

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

se lit "Dans l'environnement de typage Env, l'expression e est de type T".

Les règles de typage (1)

Constantes : Toute constante a son type :

$$\frac{n \in \mathbb{Z}}{\texttt{Env} \vdash \texttt{n} : \texttt{int}} \qquad \frac{b \in \{true, false\}}{\texttt{Env} \vdash \texttt{b} : \texttt{bool}}$$

Variables :

$$\frac{\texttt{tp(x,Env)} = \texttt{T}}{\texttt{Env} \vdash \texttt{x} : \texttt{T}}$$

où tp(x, Env) = T si (x,T) est la déclaration la plus à gauche dans Env

Vérification de types

Règles de typage

Les règles de typage (2)

Fonction :

$$\frac{\texttt{(x,A) :: Env} \vdash \texttt{e : B}}{\texttt{Env} \vdash \texttt{fun (x : A) -> e : A -> B}}$$

Application :

$$\frac{\texttt{Env} \vdash \texttt{f : A -> B} \qquad \texttt{Env} \vdash \texttt{a : A}}{\texttt{Env} \vdash \texttt{(f a) : B}}$$

Exemple : **[Méthode !]**

Vérifions le type de `(f 3)` dans l'environnement

```
Env = [(a,int);(f, int -> int)] :
```

Sommaire

Extension

Couple

Question : Quel est le type de l'expression suivante ?

```
(3, fun (x : int) -> (x=3) )
```

1. `int -> int -> bool`
2. `int * bool`
3. `int * int -> bool`
4. `int * (int -> bool)`
5. `int -> bool`

↬ Formalisé, ça donne quoi ?

Extensions

Couple

Extension n° 1 : Les couples

```
(1,2) ;;
- : int * int = (1,2)
```

```
(1, true) ;;
- : int * bool = (1,true)
```

- Types du langage étendu : ...
  + Types de paires : `T * T'`
- Expressions du langage étendu : ...
  + Paires de la forme `(e, e')` où `e` et `e'` sont des expressions.

  (ne pas confondre avec une application `(e e')` !)
- Règle de typage

  Couple :

$$\frac{\texttt{Env} \vdash \texttt{e} : \texttt{T} \qquad \texttt{Env} \vdash \texttt{e'} : \texttt{T'}}{\texttt{Env} \vdash \texttt{(e,e')} : \texttt{T * T'}}$$

Sommaire

Extensions

Définition locale

Question : Quel est le type de l'expression suivante ?

```
let x = 3 in x*x = 9
```

1. `bool`
2. `int * bool`
3. `int`
4. `int -> bool`

Extensions

Définition locale

Extension n° 2 : Let

```
let x = 3 in x + 2 ;;
- : int = 5
```

- Expressions du langage étendu : ... + `let x = e in e'`
- Règle de typage

  Let : $$\frac{\texttt{Env} \vdash \texttt{e} : \texttt{A} \qquad (\texttt{x} , \texttt{A}) :: \texttt{Env} \vdash \texttt{e'} : \texttt{B}}{\texttt{Env} \vdash \texttt{let x = e in e'} : \texttt{B}}$$

Remarques : - Les `let` globaux peuvent être vus comme des successions de `let in`.

- Le `let` est sémantiquement équivalent à l'application d'une fonction à un argument :

$$\texttt{let x = e in e'} \equiv \texttt{(fun x -> e') e}$$

Sommaire

Type somme

Rappels

Exemple : Les arbres binaires

```
type arbre_bin =
Feuille of int
| Noeud of int * arbre_bin * arbre_bin
```

Vocabulaire :

- `Feuille` et `Noeud` sont les *constructeurs* de `arbre_bin`
- `Feuille` est le constructeur de *base*
- `Noeud` est un constructeur à trois arguments récursifs

Une *Instance* de `arbre_bin` :

```
Noeud(1, Feuille 2 , Feuille 3);;
- : arbre_bin = Noeud ( 1 , Feuille 2, Feuille 3)
```

Type somme

Rappels (2)

Nous définissons généralement des fonctions sur un type somme par récursivité structurelle.

Exemple :

```
let rec somme = function
Feuille n -> n
| Noeud (n, ab1, ab2) -> n + somme ab1 + somme ab2 ;;
somme : arbre_bin -> int = <fun>
```

"En général",

- une *clause* par constructeur
- un *appel récursif* par argument *récursif*

Extensions

Types sommes (3)

Extension n° 3 : Les types sommes

- Types du langage étendu
  ...
  + Types sommes, définis par le schéma

```
type T =
C_1 of T_1
| ...
| C_n of T_n
```

- Expressions du langage étendu ...
  + Schémas de filtrage (un schéma par type inductif)

```
match e with
C_1 (x_1_1 .. x_1_n1) -> e_1
| ...
C_n (x_n_1 .. x_n_nn) -> e_n
```

Type somme

Règle de typage (1)

Comment écrire les règles de typage d'un type somme ?

Type somme et environnement de typage

```
type T =
C_1 of T_1
| ...
C_n of T_n
```

Chaque définition de type augmente l'environnement actuel avec les déclarations `(C_i , T_i -> T)`

Type somme et règle de typage

→ Pour chaque constructeur, la règle de typage est en fait la règle de typage pour les fonctions

Application (rappel) :

$$\frac{\texttt{Env} \vdash \texttt{f} : \texttt{A -> B} \qquad \texttt{Env} \vdash \texttt{a} : \texttt{A}}{\texttt{Env} \vdash \texttt{(f a)} : \texttt{B}}$$

Type somme

Exemple

Exemple : Le type

```
type arbre_bin =
 Feuille of int
 | Noeud of int * arbre_bin * arbre_bin
```

entraine la modification de l'environnement :

```
Env = [ (Noeud, int*arbre_bin*arbre_bin -> arbre_bin)
, (Feuille, int -> arbre_bin)] @ Env
```

Exemple :

Vérifier le type de l'expression `Feuille 3` dans cet environnement.

Extensions

Types sommes (4)

Pour les plus courageux...

- Règle : Typage du filtrage

$$\frac{\texttt{Env} \vdash \texttt{e} : \texttt{T} \qquad [(x_1 , T_1) , \ldots, (x_n , T_n)] \texttt{ @ Env} \vdash \texttt{e'} : \texttt{T}}{\texttt{Env} \vdash \texttt{match e with} \ldots \texttt{|}\ \texttt{C}(x_1, \ldots, x_n) \texttt{ -> e'} : \texttt{T'}}$$

où `T` est un type somme, et `C of` T_1 `*` ... `*` T_n est l'un des constructeurs de `T`.

Sommaire

Où en est-on ?

Nous avons déjà croisé,

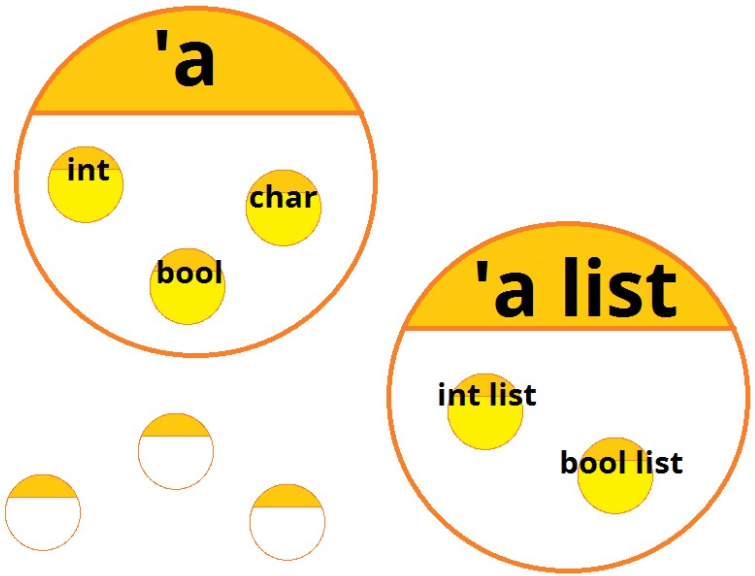
Exemple

```
let rec long = function
| x :: q -> 1 + long q
| [] -> 0 ;;
long : 'a list -> int = <fun>
```

Nous avions appelé ça du *polymorphisme*.

Polymorphisme

Type polymorphe



Un type est dit polymorphe s'il admet de multiples instances de types.

Rappel : Caml est un langage de programmation à *typage unique* (Cf. Chapitre 0)

Polymorphisme

Type polymorphe (2)

Exemple n° 1 : Les listes polymorphes

```
[true ; false ; true] ;;
- : bool list = [true;false;true]

[[1];[2];[3]];;
- : int list list = [[1];[2];[3]]
```

Attention - Les éléments doivent pourtant avoir un type uniforme :

```
[1 ; true ; [3]] ;;
Cette expression a le type bool
mais est utilisée avec le type int
```

Exemple n° 2 : `type 'a multiset` en TP n° 1

Polymorphisme

Fonction polymorphe

Une fonction polymorphe n'a pas besoin de connaître les instances de types de ses arguments.

Exemple

```
let rec long = function
[] -> 0
| x :: q -> 1 + long q ;;
long : 'a list -> int = <fun>
```

`'a list` est un type polymorphe et la fonction `long` s'applique à toutes les instances de ce type.

Polymorphisme

Un peu d'abstraction...

Un type polymorphe est comme une fonction sur des types.

En effet,

- Un type polymorphe a un ou plusieurs paramètres de type dénotés par des variables de types.

  Notation en Caml : `'a , 'b , ...` **Exemple :** `'a list`

- Une instance d'un type polymorphe a des arguments de type, possiblement imbriqués.

  Exemple :
  `[[(1,true); (2,false); (3,false)]]`
  a le type `(int * bool) list list`

Attention - Notation postfixe `int list` et non pas `list (int)`.

Extension

Polymorphisme

Exemple - La fonction de type `list`.

Question : Qui est l'intrus ?

1. `int list`
2. `int list list`
3. `int bool list`
4. `(int * bool) list`
5. `bool list`

Extension

Polymorphisme

Revenons à nos moutons, étendons notre système de types :

- Types du langage avec polymorphisme :

Types de base	`T := int	...`
Variables de type : 'a, 'b, ...	`| VT`
Types fonctionnels	`| T -> T`
Types de couples	`| T * T`
Application de fonctions de type	`| (T...T) FT`

Quelles règles de typage faut-il ajouter ?

Polymorphisme

Substitution

Avertissement : notion importante !

Définition

Une substitution

$$\sigma = [\alpha_1 \leftarrow S_1, \ldots, \alpha_n \leftarrow S_n]$$

appliquée à un type polymorphe T, remplace en parallèle toutes les occurrences de α_i par S_i dans T.

Notation : T_σ

Exemple : `('a list)['a ← bool] = bool list`

Polymorphisme

Instance et règle de typage

Définition

Un type T_i est une instance de T s'il existe une substitution σ telle que $T_i = T_\sigma$.

Exemple : `bool list` est une instance de `'a list`.

- Règle de typage (Modification de la règle d'application) :

$$\frac{\texttt{Env} \vdash \texttt{f} : A \texttt{ -> } B \qquad \texttt{Env} \vdash \texttt{a} : A_\sigma}{\texttt{Env} \vdash \texttt{(f a)} : B_\sigma}$$