

# Les types (que l'on dit simples)

Luc.Maranget@inria.fr

<http://www.enseignement.polytechnique.fr/profs/informatique/Luc.Maranget/TLP/>

- A Qu'est-ce qu'un type ?
- B PCF simplement typé.
- C Sémantique dénotationnelle.

## Quelques problèmes à l'exécution

- ▶ Par exemple, **(Fun x -> x) + 1**
- ▶ Ou bien, **1 2** (un appliqué à deux)
- ▶ Et même, **Ifz (Fun x -> x+1) Then 0 Else 1**

Et en effet,

- ▶ additionner des fonctions (impossible),
- ▶ appliquer des entiers (impossible),
- ▶ tester si une fonction vaut 0 (possible, mais bizarre, ne présage rien de bon pour la suite).

## Ce qui se passe à l'exécution

Soit le cas **(Fun x -> x)+1**.

Dans par exemple la machine ASEC.

```

let rec run a s e c = match a,s,c with
...
| Int n2,Val (Int n1)::s,IOp ADD::c ->
    run (Int (n1+n2)) s e c
...
| _,_,- -> raise Error

```

Donc : lancer une exception (erreur détectée).

Ou dans le code compilé :

```

#IOp (Add)
    addl %eax,0(%esp)
    popl %eax

```

Donc : calculer n'importe quoi (erreur non-détectée).

### Ce que dit la sémantique

De par exemple « 1 2 ».

- ▶ À grands pas : rien.
- ▶ À petits pas : terme bloqué (forme normale qui n'est pas une valeur).
- ▶ Dénotationnelle : c'est quoi déjà ?

### L'erreur peut être peu évidente

```

Let fact =
  Fix fact -> Fun n ->
    Ifz n Then 1 Else n * fact (n-1) In
Let r = Ifz 100-fact 10 Then 0 Else (Fun x -> x) In
  r 1

```

L'exécution échoue (application de 0 à 1).

On doit pouvoir détecter cette erreur à l'usine (compilation), avant qu'elle ne se produise chez le client (exécution).

### Sémantique dénotationnelle

Associer à un terme  $t$  (syntaxe) un "vrai" truc  $t = \llbracket t \rrbracket$ .

Vrai veut dire disons,

- ▶ Avec les moyens reconnus légaux (cf. Marx).
- ▶ Vrais (cf. Platon).

Simplifions : il existe  $\mathbb{N}$  et une "vraie" fonction est une fonction de  $A$  dans  $B$  où  $A$  et  $B$  sont des ensembles.

On peut donc essayer d'associer :

- ▶ À 0, l'entier ( $\in \mathbb{N}$ ) 0.
- ▶ À **Fun**  $x \rightarrow x+1$ , la fonction de  $\mathbb{N}$  dans  $\mathbb{N}$ , qui à  $x$  associe  $x + 1$ .
- ▶ À **Fun**  $x \rightarrow x$ , la fonction de ? dans ?, qui à  $x$  associe  $x$ .
- ▶ À **Fun**  $x \rightarrow x x$  ? (Là, ça va pas être possible.)

### Deux problèmes, une solution

Les problèmes :

- ▶ Ne livrer que des programmes sans erreurs (sans erreurs du style « 1 2 »).
- ▶ Définir la sémantique dénotationnelle de PCF.
- ▶ (Éviter les paradoxes du genre  $X = \{x \mid x \notin x\}$ .)

La solution : obliger le programmeur à spécifier le domaine de définition des fonctions.

Une solution radicale, mais qui va fonctionner.

### Un nouveau PCF

```

t ::= n
    | x
    | t1 op t2
    | t1 t2
    | Fun (x:T) -> t
    | Fix (x:T) -> t
    | Let x = t1 In t2

```

Cf. définition de méthode en Java:

```
int f(int x) { ... }
```

### La même chose en Caml

Syntaxe abstraite du nouveau PCF (dans T.Ast).

```

(*
   Var.t   : type des variables (string)
   T.Type.t : type des types (???)
*)
type t =
  | Num of int
  | Var of Var.t
  | Op  of Op.t * t * t
  | Ifz of t * t * t
  | Let of Var.t * t * t
  | App of t * t
  | Fun of Var.t * T.Type.t * t
  | Fix of Var.t * T.Type.t * t

```

Mais qu'est-ce que T.Type.t (T) ?

### Ensembles de définition des fonctions

Pour éviter « 1 2 » et « (Fun x -> x+1)+2 », il doit suffire de séparer entiers et fonctions.

Pour autoriser

```
Let f = Fun x -> x+1 In f 1 + 2
```

Et interdire

```
Let f = Fun x -> x 1 In f 1 + 2
```

Il faut être un peu plus détaillé, les deux types de f

```
Nat -> Nat           (Nat -> Nat) -> Nat
```

### Algèbre des types

Les types sont décrits ainsi :

- ▶ Nat est un type.
- ▶ Si A et B sont des types, alors A -> B est un type.

Autrement dit :

$$\mathbb{N} \in \mathcal{T} \qquad \frac{A \in \mathcal{T} \quad B \in \mathcal{T}}{A \rightarrow B \in \mathcal{T}}$$

Ou encore :

```
type t = Nat | Arrow of t * t
```

C'est le langage des types (simples), il n'a pour le moment aucune sémantique.

## Types et ensembles

Idée évidente pour la sémantique : des ensembles.

Selon notre programme de vérifier les types à la compilation, les types ne peuvent pas être des ensembles quelconques.

**Ex.** Pourquoi pas un type  $\mathbb{N}^*$  ? ( $\text{Nat}^*$ ) Ça serait bien commode :

**Let** `div = Fun (x:Nat) -> Fun (y:Nat*) -> x / y`

Et bien non,

► Typer `div 100 (fact 10-100)` demande de prouver que la valeur de `fact 10` est  $> 100$ , c'est beaucoup demander à un compilateur.

► Quel type donner à la soustraction ?

Tant pis : l'usine (compilateur) peut livrer des programmes qui divisent par zéro.

## Contrôle à l'usine

Écrire **Fun**  $(x : A) \rightarrow t$  n'empêche pas les erreurs.

**(Fun (x:Nat) -> x 1) 2**

Mais cela permet de définir facilement un sous-ensemble des termes : les termes bien typés.

L'ensemble des termes bien typés est défini par une relation

$$E \vdash t : A$$

(Le terme  $t$  possède le type  $A$  dans l'environnement  $E$ .)

Dans le cas d'un environnement vide on abrège

$$t : A$$

(Le terme  $t$  a le type  $A$ .)

## Exemples

**Fun**  $(x:\text{Nat}) \rightarrow x \ 1$

Mal typé, car  $x$  de type  $\text{Nat}$  est appliqué (à 2).

**Fix**  $(x:\text{Nat}) \rightarrow x + 1$

Bien typé.

**Fix**  $(f:\text{Nat}) \rightarrow$   
**Fun**  $(n:\text{Nat}) \rightarrow$   
**Ifz**  $n$  **Then** 1 **Else**  $2 * f (n-1)$

Mal typé, car  $f$  de type  $\text{Nat}$  est appliqué (à  $n-1$ ). On aurait pu écrire:

**Fix**  $(f:\text{Nat} \rightarrow \text{Nat}) \rightarrow$   
**Fun**  $(n:\text{Nat}) \rightarrow$   
**Ifz**  $n$  **Then** 1 **Else**  $2 * f (n-1)$

## Définition inductive de $E \vdash t : A$

$$E \vdash n : \text{Nat} \quad \frac{E(x) = A}{E \vdash x : A} \quad \frac{E \vdash t_1 : \text{Nat} \quad E \vdash t_2 : \text{Nat}}{E \vdash t_1 \text{ op } t_2 : \text{Nat}}$$

$$\frac{E \vdash t_1 : \text{Nat} \quad E \vdash t_2 : A \quad E \vdash t_3 : A}{E \vdash \text{Ifz } t_1 \text{ Then } t_2 \text{ Else } t_3 : A}$$

$$\frac{E \vdash t_1 : A \rightarrow B \quad E \vdash t_2 : A}{E \vdash t_1 t_2 : B} \quad \frac{E \oplus [x : A] \vdash t : B}{E \vdash (\text{Fun } (x : A) \rightarrow t) : A \rightarrow B}$$

$$\frac{E \oplus [x : A] \vdash t : A}{E \vdash (\text{Fix } (x : A) \rightarrow t) : A}$$

$$\frac{E \vdash t_1 : A \quad E \oplus [x : A] \vdash t_2 : B}{E \vdash \text{Let } x = t_1 \text{ In } t_2 : B}$$

## Typage de la puissance de deux

$$\frac{\frac{[\dots; n : \text{Nat}] \vdash n : \text{Nat} \quad E \vdash 1 : \text{Nat} \quad E \vdash t_2 : \text{Nat}}{[f : \text{Nat} \rightarrow \text{Nat}, n : \text{Nat}] \vdash \text{Ifz } n \text{ Then } 1 \text{ Else } t_2 : \text{Nat}}}{[f : \text{Nat} \rightarrow \text{Nat}] \vdash (\text{Fun } (x : \text{Nat}) \rightarrow \text{Ifz } \dots) : \text{Nat} \rightarrow \text{Nat}}}{\frac{\emptyset \vdash (\text{Fix } (f : \text{Nat} \rightarrow \text{Nat}) \rightarrow t) : \text{Nat} \rightarrow \text{Nat} \quad \emptyset \vdash 100 : \text{Nat}}{\emptyset \vdash (\text{Fix } (f : \text{Nat} \rightarrow \text{Nat}) \rightarrow t) 100 : \text{Nat}}}$$

Avec  $E \vdash t_2 : \text{Nat}$  prouvé comme suit :

$$\frac{E \vdash 2 : \text{Nat} \quad \frac{E \vdash f : \text{Nat} \rightarrow \text{Nat} \quad \frac{E \vdash n : \text{Nat} \quad E \vdash 1 : \text{Nat}}{E \vdash n-1 : \text{Nat}}}{E \vdash f (n-1) : \text{Nat}}}{[f : \text{Nat} \rightarrow \text{Nat}, n : \text{Nat}] \vdash 2 * f (n-1) : \text{Nat}}$$

**N.B.** Inutile de calculer  $2^{100}$ .

## La relation $E \vdash t : A$

Ressemble très fortement à une sémantique « à grand pas ».

Pour cela, on l'appelle :

- ▶ Sémantique *statique* (car valable au moment de la compilation).
- ▶ Interprétation *abstraite*, car elle est une *abstraction* de la sémantique, qui calcule des ensembles de valeurs et non plus des valeurs.

Calculer  $A$  tel que  $t : A$  est surtout *vérifier* le type de  $t$ , car on ne découvre (infère) pas franchement  $A$ .

## Décidabilité

Avec des règles aussi précises,

- ▶ On écrit facilement un programme qui étant donnés  $E$  et  $t$  trouve  $A$  tel que  $E \vdash t : A$  ou échoue.
- ▶ Et de plus  $A$  est unique.
- ▶ On dit que la relation «  $\emptyset \vdash t : A$  », notée «  $t : A$  », est décidable et non-ambigüe.

C'est ce que nous ferons en TP.

```
open T
exception Error
```

```
val check : (Var.t * Type.t) list -> Ast.t -> Type.t
(* L'appel 'check E t' envoie A tel que E ⊢ t : A,
   ou lance Error si il n'existe pas de tel A *)
```

## Correction du typage statique

On veut exprimer que si un terme est bien typé (à la compilation), alors il ne se produit pas d'erreurs (de type) lors de l'exécution.

- ▶ Sémantique à petit pas. Deux gros lemmes :
  - ▷ *subject reduction*, c-à-d, si  $t : A$  et  $t \rightarrow t'$ , alors  $t' : A$ . Cela se montre par induction sur la dérivation  $t : A$ .
  - ▷ *progress*, c'-à-d, si  $t : A$  et  $t$  n'est pas une valeur, alors  $t$  n'est pas en forme normale. Cela se montre en détaillant les termes bloqués et en montrant qu'ils ne sont pas typables.
- ▷ Il vient ce **Théorème** (*Correction du typage statique*),  $t : A$  entraîne l'absence d'erreurs (de type) à l'exécution.

## Correction, à grand pas

Une méthode pas satisfaisante, prouver:

$$(t : A \wedge t \hookrightarrow v) \Rightarrow v : A$$

Un peu court, car termes qui bouclent et erreurs de type sont confondus.

On ajoute une « valeur » **wrong** qui exprime une erreur de type, et des règles de propagation.

$$\frac{t_1 \hookrightarrow n}{t_1 t_2 \hookrightarrow \text{wrong}} \quad \frac{t_1 \hookrightarrow \text{wrong}}{t_1 t_2 \hookrightarrow \text{wrong}} \quad \frac{t_2 \hookrightarrow \text{wrong}}{t_1 t_2 \hookrightarrow \text{wrong}} \quad \dots$$

On obtient alors une formulation plus précise, qui entraîne:

$$(t : A \wedge t \hookrightarrow v) \Rightarrow v \neq \text{wrong}$$

## Une dernière réflexion

Le typage  $A : t$  garantit l'absence d'erreurs de type à l'exécution.

La réciproque est bien entendu fausse.

**(Ifz 0 Then 0 Else (Fun x -> x))+1**

Cette *incomplétude* est un prix à payer pour le le typage statique.

Pourquoi ? À cause de l'indécidabilité de l'arrêt : il n'existe pas d'algorithme qui permet de décider si  $P$  (bien typé) termine.

Soit  $P$  bien typé (et donc exempt d'erreur) et :

**(Ifz P Then 0 Else (Fun x -> x)) + 1**

Un erreur se produit à l'exécution entraîne  $P$  termine (et  $P$  s'évalue  $n$  différent de zéro).

## Un dernier théorème

Si  $t : A$  et que  $t$  ne contient pas **Fix**, alors toutes les réductions issues de  $t$  terminent (Tait).

**Corollaire :**

- ▶ Le terme **(Fun x -> x x) (Fun x -> x x)** n'est pas typable.
- ▶ Aucun codage du point fixe n'est typable. Disons qu'un codage du pt. fixe est un terme  $Y$  avec

$$Y t \rightarrow^* t (Y t)$$

On peut alors coder **Fix**  $f \rightarrow t$  comme  $Y$  **(Fun**  $f \rightarrow t$ **)**.

Pour un exemple, cf poly (Exercice 2.10).

## Sémantique dénotationnelle

Maintenant que les termes contiennent des types, il devient possible d'associer une fonction des mathématiques à toute fonction PCF.

Commençons par associer (fonction  $[[\cdot]]$ ) un ensembles à chaque type de PCF.

$$[[\text{Nat}]] = \mathbb{N} \quad [[A \rightarrow B]] = [[A]] \rightarrow [[B]]$$

(où  $E_1 \rightarrow E_2$  est l'ensemble des fonctions de  $E_1$  vers  $E_2$ ).

On a alors six règles évidentes.

$$[[n]] = n \quad [[t_1 \text{ op } t_2]] = [[t_1]] \text{ op } [[t_2]]$$

$$[[\text{Ifz } t_1 \text{ Then } t_2 \text{ Else } t_3]] =$$

$$[[t_2]] \text{ si } [[t_1]] \text{ est } 0, \text{ ou } [[t_3]] \text{ si } [[t_1]] \text{ est } n \neq 0$$

## Sémantique dénotationnelle triviale

À  $t : A$  on associe  $\llbracket t \rrbracket$  dans  $\llbracket A \rrbracket$ . ( $\llbracket t \rrbracket$  est une *valeur sémantique*, un élément de  $\bigcup_{A \in \mathcal{T}} \llbracket A \rrbracket$ ).

La sémantique des termes ouverts est relative à un *environnement sémantique*  $E$  des variables vers les valeurs sémantiques,

- ▶  $\llbracket x \rrbracket_E = E(x)$
- ▶  $\llbracket \mathbf{Let} \ x = t_1 \ \mathbf{In} \ t_2 \rrbracket_E = \llbracket t_2 \rrbracket_{E \oplus [x = \llbracket t_1 \rrbracket_E]}$
- ▶  $\llbracket \mathbf{Fun} \ (x : A) \rightarrow t \rrbracket_E$ , la fonction qui à tout  $v$  de  $\llbracket A \rrbracket$ , associe  $\llbracket t \rrbracket_{E \oplus [x = v]}$ .
- ▶  $\llbracket t_1 \ t_2 \rrbracket_E = \llbracket t_1 \rrbracket_E(\llbracket t_2 \rrbracket_E)$

Avec une définition à peine modifiée (qui explicite le type des valeurs sémantiques de l'environnement), on verrait que si  $t : A$ , alors  $\llbracket t \rrbracket$  existe et  $\in \llbracket A \rrbracket$ , mais bon.

## Frustration

Toute cette dépense théorique pour des paraphrases en italiques !

Mais, la sémantique triviale laisse de côté deux questions.

- ▶ Que vaut  $\llbracket 1/0 \rrbracket$  ? Faux problème, cela vaut *error*, on décide  $\mathit{error} \in \llbracket A \rrbracket$  pour tout  $A$ , et on modifie les règles pour propager cette erreur.
- ▶ Et le point fixe ? Vrai problème.

La tentation est grande de définir  $\llbracket \mathbf{Fix} \ (x : A) \rightarrow t \rrbracket$  comme point fixe de

$$\llbracket \mathbf{Fun} \ (x : A) \rightarrow t \rrbracket$$

Mais cela ne va pas se faire tout seul.

## Point fixe

En PCF on peut écrire le terme  $\mathbf{Fix} \ (x : A) \rightarrow t$  (du moment que l'on a  $[x : A] \vdash t : A$ ).

On se se préoccupe pas de l'existence du point fixe de  $\llbracket \mathbf{Fun} \ (x : A) \rightarrow t \rrbracket$ . Par exemple :

$$T_1 = \mathbf{Fix} \ (x : \mathbf{Nat}) \rightarrow x + 1$$

Or, il n'existe pas d'entier  $x$  tel que  $x = x + 1$ .

Si on revient à la sémantique opérationnelle, on avait :

$$(\mathbf{Fix} \ (x : A) \rightarrow t) \rightarrow [\mathbf{Fix} \ (x : A) \rightarrow t \setminus x]t$$

Et donc :

$$T_1 \rightarrow T_1 + 1 \rightarrow (T_1 + 1) + 1 \rightarrow \dots$$

Pour compléter les valeurs sémantiques on donne la valeur  $\perp$  à ce terme (terme de type  $\mathbf{Nat}$ , qui ne termine pas).

## Une vérification

On a donc posé  $\llbracket \mathbf{Nat} \rrbracket = \mathbb{N} \cup \{\perp\}$ .

Si on complète les quatres règles des opérations

$$\llbracket t_1 \ \mathit{op} \ t_2 \rrbracket = \perp, \text{ si } \llbracket t_1 \rrbracket \text{ ou } \llbracket t_2 \rrbracket \text{ est } \perp$$

Alors  $\perp$  est l'unique point fixe de  $\llbracket \mathbf{Fun} \ (x : \mathbf{Nat}) \rightarrow x + 1 \rrbracket$

De même,  $\llbracket \mathbf{Fun} \ (x : \mathbf{Nat}) \rightarrow x \rrbracket$  (fonction qui à  $x$  entier associe  $x$ ) avait plusieurs point fixes  $0, 1, 2, \dots$

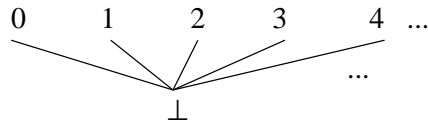
Maintenant, cette fonction (qui à  $x \in \llbracket \mathbf{Nat} \rrbracket$  associe  $x$ ) a un pt. fixe de plus  $\perp$ .

Et c'est celui là que l'on veut pour  $\llbracket \mathbf{Fix} \ (x : \mathbf{Nat}) \rightarrow x \rrbracket$  (termine pas).

Points fixes de  $\llbracket \mathbf{Fun} \ (x : \mathbf{Nat}) \rightarrow x + x \rrbracket$  ?  $\perp$  et  $0$ . Valeur de  $\llbracket \mathbf{Fix} \ (x : \mathbf{Nat}) \rightarrow x + x \rrbracket$  ?  $\perp$

### Comment choisir son point fixe

Ordre de Scott sur le domaine  $\llbracket \text{Nat} \rrbracket$ .



Ordre faiblement complet ? Oui !

On veut le plus petit point fixe.

Il existe bien pour toutes les fonctions croissantes et continues de  $\llbracket \text{Nat} \rrbracket$  dans  $\llbracket \text{Nat} \rrbracket$ .

Ce qui est le cas de  $\llbracket \text{Fun } (x:\text{Nat}) \rightarrow x+1 \rrbracket$  etc.

### Et dans le cas général...

Définition revue des domaines sémantiques :

- ▶  $\llbracket \text{Nat} \rrbracket = \mathbb{N} \cup \{\perp\}$
- ▶  $\llbracket A \rightarrow B \rrbracket$  fonctions *croissantes et continues* de  $\llbracket A \rrbracket$  dans  $\llbracket B \rrbracket$ .

L'ordre sur  $\llbracket A \rightarrow B \rrbracket$  est défini point à point,  $f \leq g$  ssi  $\forall x \in \llbracket A \rrbracket, f(x) \leq g(x)$ .

Il y a un plus petit elt : la fonction qui à  $x \in \llbracket A \rrbracket$  associe  $\perp_B$  (notée  $\perp_A \rightarrow B$ ).

Cette relation est faiblement complète (permutation de limites).

Tous les termes de type  $A$  s'interprètent bien dans  $\llbracket A \rrbracket$  (continuité).

La valeur de  $\text{Fix } (x : A) \rightarrow t$  est *définie* comme le plus petit point fixe de  $\llbracket \text{Fun } (x : A) \rightarrow t \rrbracket$ .

### Qu'est-ce que ça veut dire en pratique ?

Fonction puissance de deux :

```

Let P =
  Fix (p:Nat -> Nat) ->
    Fun (n:Nat) ->
      Ifz n Then 1 Else 2 * p (n-1)

```

Noté  $\text{Fix } (p:\text{Nat} \rightarrow \text{Nat}) \rightarrow t$ .

Pour avoir la valeur du terme ci-dessus, on calcule le pppf de  $\Phi = \llbracket \text{Fun } (p:\text{Nat}) \rightarrow t \rrbracket$ .

Le plus pppf de  $\Phi$  est la limite de la suite  $(a_k) = (\Phi^k(\perp_{\text{Nat}} \rightarrow \text{Nat}))$ .

$a_0$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	...
$a_1$	$\perp$	1	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	...
$a_2$	$\perp$	1	2	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	...
$a_3$	$\perp$	1	2	4	$\perp$	$\perp$	$\perp$	$\perp$	...
$a_4$	$\perp$	1	2	4	8	$\perp$	$\perp$	$\perp$	...
$a_5$	$\perp$	1	2	4	8	16	$\perp$	$\perp$	...
$a_6$	$\perp$	1	2	4	8	16	32	$\perp$	...

Car par exemple :  $a_6(5) = 2 * a_5(4)$  (en sautant quelques étapes).



### Mise en relation des sémantiques

Il y a un théorème (pas facile) :

$$\llbracket t \rrbracket = n \Leftrightarrow t \hookrightarrow n$$

Ce théorème s'applique-t-il à l'évaluation en appel par nom ou par valeur ?

Considérer :

```
Let k = Fun (x:Nat) -> Fun (y:Nat) -> y In
Let loop = Fix (n:Nat) -> n In
k 0 loop
```

Cela vaut  $\llbracket y \rrbracket_{[loop=\perp, y=0]}$  qui vaut 0.

Donc c'est appel par nom.

### À quoi ça sert ?

C'est beau de savoir que l'on programme de vraies fonctions.

À prouver des théorèmes. Par ex.

On ne peut pas programmer en PCF typé, la fonction  $g$  qui prend  $f : \text{Nat} \rightarrow \text{Nat}$  en argument, et renvoie 0 si  $f$  0 vaut zéro et 1 autrement.

Soit donc  $g$  de  $\llbracket (\text{Nat} \rightarrow \text{Nat}) \rightarrow \text{Nat} \rrbracket$  telle que ci-dessus.

Considérer  $f_0 = \llbracket \text{Fun } x:\text{Nat} \rightarrow 0 \rrbracket$ ,  $f_1 = \llbracket \text{Fun } x:\text{Nat} \rightarrow 1 \rrbracket$ , On  $\perp_{\text{Nat} \rightarrow \text{Nat}} \leq f_0$  et  $\perp_{\text{Nat} \rightarrow \text{Nat}} \leq f_1$ . Soit ( $g$  croissante)  $g(\perp_{\text{Nat} \rightarrow \text{Nat}}) = \perp$  C.Q.F.D.

Bon on joue sur les mots, voir Exercice 5.11 pour un exemple plus sérieux.

► TP, vérification de type.

► La prochaine fois, inférence de type.