

Cours 6

Logique et Informatique

Preuves et Programmes

Logique – Licence Informatique



Preuves = Programmes

- écrire un programme ... c'est écrire une preuve
- un mini-langage de programmation
 - ▶ symboles de variables : x, y, z, \dots
 - ▶ définitions de fonctions : $\text{function } x \mapsto e$
 - ▶ application d'une fonction e_1 à un argument e_2 : $(e_1 \ e_2)$
 - ▶ exemples
 - ★ fonction identité
 $\text{function } x \mapsto x$
 - ★ application de la fonction identité à elle même
 $(\text{function } x \mapsto x \ \text{function } x \mapsto x)$
 - ★ (opérateur de) composition de fonction
 $\text{function } f \mapsto \text{function } g \mapsto \text{function } x \mapsto (f \ (g \ x))$

OCaml

```
# let f1 = function x -> x+1;;
# let f2 = function x -> x*x;;
# let compose = function f -> function g -> function x -> (f (g x));;
# let f = (compose f1 f2);;
# (f 4);;
- : int = 17
```

Typage des programmes

- associer un type à chaque programme « typable »
- langage de types
 - ▶ types de base τ_1, τ_2, \dots
 - ▶ type des fonctions : si A et B sont des types alors $A \rightarrow B$ est un type (c'est le type des fonctions qui étant donné un argument de type A retourne un résultat de type B)
 - ▶ exemples
 - ★ la fonction identité `function x ↦ x` est de type $A \rightarrow A$ (où A est « n'importe quel type »)
 - ★ la fonction qui permet de composer 2 fonctions

`function f ↦ function g ↦ function x ↦ (f (g x))`

est de type : $(A \rightarrow B) \rightarrow ((C \rightarrow A) \rightarrow (C \rightarrow B))$
 (où A, B, C sont « n'importe quels types »)

OCaml

```
# function x -> x;;
- : 'a -> 'a = <fun>
# function f -> function g -> function x -> (f (g x));;
- : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>
```

Typage des programmes

- associer un type à chaque programme « typable »
 - ▶ règles de typage des programmes
 - ▶ typage statique : le type des programmes est calculé avant l'exécution (OCaml, etc.)
 - ★ par d'erreur de type à l'exécution, les programmes « mal typés » sont rejetés
 - ▶ typage dynamique : les types sont déterminés à l'exécution (Python, etc.)
 - ★ plus de programmes acceptés mais c'est au programmeur de s'assurer que son code ne provoquera pas d'erreur de type à l'exécution
- environnement de typage : permet d'associer des types aux variables
 - ▶ ensemble de liaisons de typage de la forme (x, A)
 - ★ le typage (x, A) exprime que la variable x est de type A

Typage des variables

ce que l'on cherche à typer est présent dans l'environnement de typage

$\langle i \rangle$ dans l'environnement $h_1 : (x_1, A_1), \dots, h_n : (x_n, A_n), h : (x, A)$
 typons le programme x avec le type A
 $\langle i \rangle$ CQFD (Var avec h)

$h : (x, A)$ peut également être présent dans une boîte $\langle j \rangle$ dont la boîte $\langle i \rangle$ est une sous-boîte

$\langle j \rangle$ dans l'environnement $h'_1 : (x'_1, A'_1), \dots, h'_k : (x'_k, A'_k), h : (x, A)$
 ...

$\langle i \rangle$ dans l'environnement $h_1 : (x_1, A_1), \dots, h_n : (x_n, A_n)$
 typons le programme x avec le type A
 $\langle i \rangle$ CQFD (Var avec h)

 ...
 $\langle j \rangle$ CQFD (Nom)

Typage des fonctions

pour pouvoir typer la fonction `function $x \mapsto e$` avec le type $A \rightarrow B$, il suffit de pouvoir typer `e` avec le type B dans un environnement de typage où le type de `x` est A

$\langle i \rangle$ dans l'environnement $h_1 : (x_1, A_1), \dots, h_n : (x_n, A_n)$
 typons le programme `function $x \mapsto e$` avec le type $A \rightarrow B$

$\langle i + 1 \rangle$ dans l'environnement $h : (x, A)$
 typons le programme `e` avec le type B

...

$\langle i + 1 \rangle$ CQFD

$\langle i \rangle$ CQFD (*Fun*)

Typage des applications de fonction

si e_1 est une fonction de type $A \rightarrow B$ et si e_2 est de type A , alors l'application de $(e_1 \ e_2)$ de e_1 à e_2 est de type B

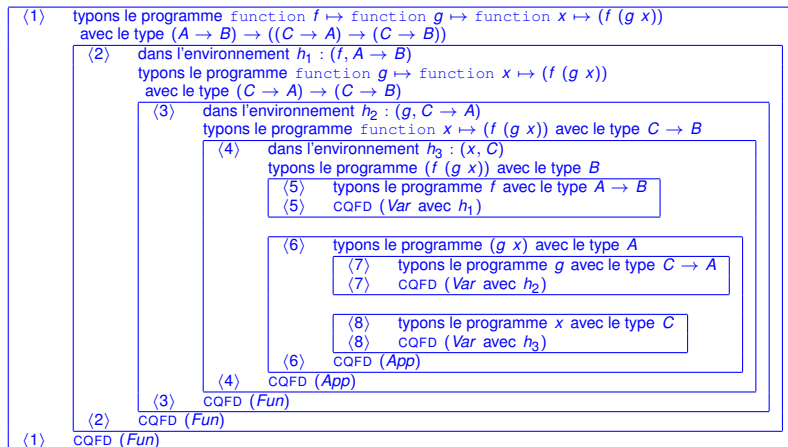
$\langle i \rangle$ dans l'environnement $h_1 : (x_1, A_1), \dots, h_n : (x_n, A_n)$
 typons le programme $(e_1 \ e_2)$ avec le type B

$\langle i + 1 \rangle$ typons le programme e_1 avec le type $A \rightarrow B$
 \dots
 $\langle i + 1 \rangle$ CQFD

$\langle i + 2 \rangle$ typons le programme e_2 avec le type A
 \dots
 $\langle i + 2 \rangle$ CQFD

$\langle i \rangle$ CQFD (*App*)

Exemple



Typage et déduction

$\langle i \rangle$ dans l'environnement $h_1 : (x_1, A_1), \dots, h_n : (x_n, A_n), h : (x, A)$
 typons le programme x avec le type A
 $\langle i \rangle$ CQFD (Var avec h)

$\langle i \rangle$ supposons $h_1 : A_1, \dots, h_n : A_n, h : A$
 montrons A
 $\langle i \rangle$ CQFD (Ax avec h)

Typage et déduction

$\langle i \rangle$ dans l'environnement $h_1 : (x_1, A_1), \dots, h_n : (x_n, A_n)$
 typons le programme `function $x \mapsto e$` avec le type $A \rightarrow B$

$\langle i + 1 \rangle$ dans l'environnement $h : (x, A)$
 typons le programme `e` avec le type B

...

$\langle i + 1 \rangle$ CQFD

$\langle i \rangle$ CQFD (*Fun*)

$\langle i \rangle$ supposons $h_1 : A_1, \dots, h_n : A_n$
 montrons $A \Rightarrow B$

$\langle i + 1 \rangle$ supposons $h : A$
 montrons B

...

$\langle i + 1 \rangle$ CQFD

$\langle i \rangle$ CQFD ($I \Rightarrow$)

Typage et déduction

$\langle i \rangle$ dans l'environnement $h_1 : (x_1, A_1), \dots, h_n : (x_n, A_n)$
 typons le programme $(e_1 \ e_2)$ avec le type B

$\langle i + 1 \rangle$ typons le programme e_1 avec le type $A \rightarrow B$

...

$\langle i + 1 \rangle$ CQFD

$\langle i + 2 \rangle$ typons le programme e_2 avec le type A

...

$\langle i + 2 \rangle$ CQFD

$\langle i \rangle$ CQFD (*App*)

$\langle i \rangle$ supposons $h_1 : A_1, \dots, h_n : A_n$, montrons B

$\langle i + 1 \rangle$ montrons $A \Rightarrow B$

...

$\langle i + 1 \rangle$ CQFD

$\langle i + 2 \rangle$ montrons A

...

$\langle i + 2 \rangle$ CQFD

$\langle i \rangle$ CQFD (E_{\Rightarrow})

Correspondance de Curry-Howard

preuve	\equiv	programme
formule	\equiv	type

- les règles de typage correspondent exactement aux règles de déduction « décorées » avec des programmes
 - ▶ typer un programme c'est déterminer de quelle formule le programme est une preuve
- en « décorant » la preuve d'une formule avec des programmes on obtient le typage d'un programme
 - ▶ prouver une formule c'est écrire un programme dont le type est cette formule

Sémantique de Brouwer-Heyting-Kolmogorov

- une formule logique F n'est plus associée à une valeur de vérité ... mais au type $\mathbf{P}(F)$ de ses preuves (programmes)
 - ▶ $\mathbf{P}(A)$ est le type des preuves de A (i.e. des programmes qui prouvent A)
 - ▶ une preuve de $A \Rightarrow B$ est un programme qui associe une preuve de B à toute preuve de A

$$\mathbf{P}(A \Rightarrow B) = \mathbf{P}(A) \rightarrow \mathbf{P}(B)$$

★ exemple : la fonction identité est une preuve de $A \Rightarrow A$

- **Proposition.**

- ▶ F est une formule prouvable si et seulement si il existe un programme de type $\mathbf{P}(F)$

Types produits / Conjonctions

- on ajoute au mini-langage de programmation les couples
 - si e_1 et e_2 sont des programmes, alors (e_1, e_2) est un programme
 - projections
 - si e est un programme, alors $\text{fst}(e)$ est un programme
($\text{fst}(e_1, e_2) = e_1$)
 - si e est un programme, alors $\text{snd}(e)$ est un programme
($\text{snd}(e_1, e_2) = e_2$)
- on ajoute au langage de type les types produits
 - si A et B sont des types alors $A \times B$ est un type
 - c'est le type des couples (e_1, e_2) où e_1 est de type A et e_2 est de type B
- une preuve de $A \wedge B$ est un couple constitué d'une preuve de A et d'une preuve de B

$$\mathbf{P}(A \wedge B) = \mathbf{P}(A) \times \mathbf{P}(B)$$

OCaml

```
# (3+2,true && false);;
- : int * bool = (5, false)
# (fst (3+2,true && false));;
- : int = 5
# (snd (3+2,true && false));;
- : bool = false
```

Typage des couples

$\langle i \rangle$ dans l'environnement $h_1 : (x_1, A_1), \dots, h_n : (x_n, A_n)$
 typons le programme (e_1, e_2) avec le type $A \times B$

$\langle i + 1 \rangle$ typons le programme e_1 avec le type A

...

$\langle i + 1 \rangle$ CQFD

$\langle i + 2 \rangle$ typons le programme e_2 avec le type B

...

$\langle i + 2 \rangle$ CQFD

$\langle i \rangle$ CQFD (*Pair*)

$\langle i \rangle$ supposons $h_1 : A_1, \dots, h_n : A_n$, montrons $A \wedge B$

$\langle i + 1 \rangle$ montrons A

...

$\langle i + 1 \rangle$ CQFD

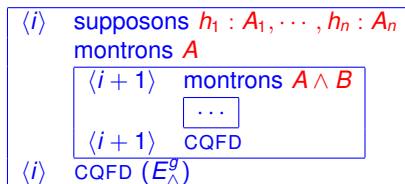
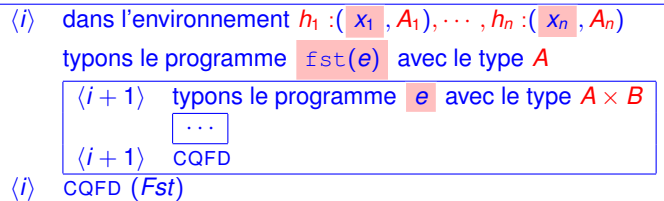
$\langle i + 2 \rangle$ montrons B

...

$\langle i + 2 \rangle$ CQFD

$\langle i \rangle$ CQFD (I_\wedge)

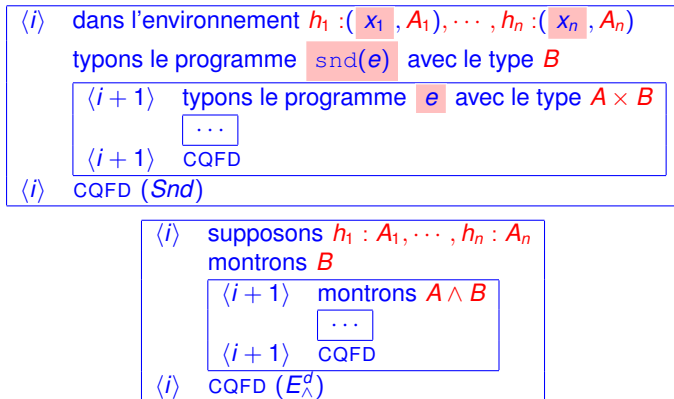
Typage des projections



OCaml

```
# fst;;
- : 'a * 'b -> 'a = <fun>
```

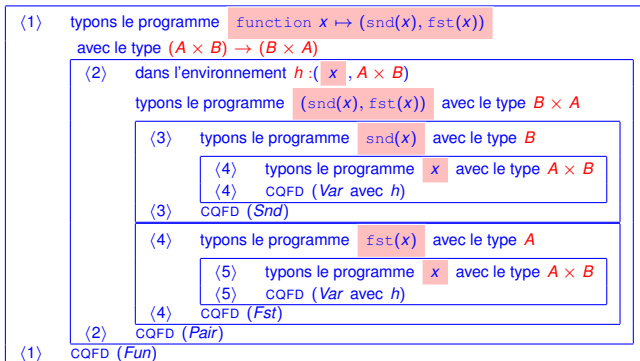
Typage des projections



OCaml

```
# snd;;
- : 'a * 'b -> 'b = <fun>
```

Exemple : $(A \wedge B) \Rightarrow (B \wedge A)$



OCaml

```
# function x -> ((snd x), (fst x));;
- : 'a * 'b -> 'b * 'a = <fun>
```

Types sommes / Disjonctions

- on ajoute au langage de type les types sommes
 - ▶ si A et B sont des types alors $A + B$ est un type
 - ▶ un programme de type $A + B$ est :
 - ★ soit $\text{inj}_g(e)$ où e est de type A
 - ★ soit $\text{inj}_d(e)$ où e est de type B

OCaml

```
# type bool_char = Inj_bool of bool | Inj_char of char;;

# Inj_bool(true);;
- : bool_char = Inj_bool true
# Inj_bool(false);;
- : bool_char = Inj_bool false
# Inj_char('a');;
- : bool_char = Inj_char 'a'
```

Types sommes / Disjonctions

- on ajoute au mini-langage de programmation
 - ▶ les « injections »
 - ★ si e est un programme, alors $\text{inj}_g(e)$ est un programme
 - ★ si e est un programme, alors $\text{inj}_d(e)$ est un programme
 - ▶ le « filtrage »
 - ★ si e, e_g, e_d sont des programmes et x_g, x_d sont des variables, alors $\text{match } e \text{ with } \text{inj}_g(x_g) \mapsto e_g \mid \text{inj}_d(x_d) \mapsto e_d$ est un programme

OCaml

```
# type bool_char = Inj_bool of bool | Inj_char of char;;
# let f = function x -> match x with
    Inj_bool(x1) -> if x1 then 1 else 0
  | Inj_char(x2) -> code(x2);;
val f : bool_char -> int = <fun>
# (f (Inj_bool(false && true)));;
- : int = 0
# (f (Inj_char('a')));;
- : int = 97
```

- une preuve de $A \vee B$ est soit une preuve de A soit une preuve de B

$$\mathbf{P}(A \vee B) = \mathbf{P}(A) + \mathbf{P}(B)$$

Typage des injections

$\langle i \rangle$ dans l'environnement $h_1 : (x_1, A_1), \dots, h_n : (x_n, A_n)$
 typons le programme $\text{inj}_g(e)$ avec le type $A + B$

$\langle i + 1 \rangle$ typons le programme e avec le type A
 \dots
 $\langle i + 1 \rangle$ CQFD

$\langle i \rangle$ CQFD (Injg)

$\langle i \rangle$ supposons $h_1 : A_1, \dots, h_n : A_n$
 montrons $A \vee B$

$\langle i + 1 \rangle$ montrons A
 \dots
 $\langle i + 1 \rangle$ CQFD

$\langle i \rangle$ CQFD (I_{\vee}^g)

Typage des injections

$\langle i \rangle$ dans l'environnement $h_1 : (x_1, A_1), \dots, h_n : (x_n, A_n)$
 typons le programme $\text{inj}_d(e)$ avec le type $A + B$

$\langle i + 1 \rangle$ typons le programme e avec le type B

...

$\langle i + 1 \rangle$ CQFD

$\langle i \rangle$ CQFD (Injd)

$\langle i \rangle$ supposons $h_1 : A_1, \dots, h_n : A_n$
 montrons $A \vee B$

$\langle i + 1 \rangle$ montrons B

...

$\langle i + 1 \rangle$ CQFD

$\langle i \rangle$ CQFD (I^d_{\vee})

Typage du filtrage

$\langle i \rangle$ dans l'environnement $h_1 : (x_1, A_1), \dots, h_n : (x_n, A_n)$
typons le programme

`match e with injg(xg) ↦ eg | injd(xd) ↦ ed` avec le type C

$\langle i+1 \rangle$ typons le programme `e` avec le type $A + B$

...

$\langle i+1 \rangle$ CQFD

$\langle i+2 \rangle$ dans l'environnement $h_A : (x_g, A)$

typons le programme `eg` avec le type C

...

$\langle i+2 \rangle$ CQFD

$\langle i+3 \rangle$ dans l'environnement $h_B : (x_d, B)$

typons le programme `ed` avec le type C

...

$\langle i+3 \rangle$ CQFD

$\langle i \rangle$ CQFD (Match)

$\langle i \rangle$ supposons $h_1 : A_1, \dots, h_n : A_n$
montrons C

$\langle i+1 \rangle$ montrons $A \vee B$

...

$\langle i+1 \rangle$ CQFD

$\langle i+2 \rangle$ supposons $h_A : A$
montrons C

...

$\langle i+2 \rangle$ CQFD

$\langle i+3 \rangle$ supposons $h_B : B$
montrons C

...

$\langle i+3 \rangle$ CQFD

$\langle i \rangle$ CQFD (E_{\vee})

Exemple : $(A \vee B) \Rightarrow (B \vee A)$

⟨1⟩ typons le programme `function x ↦ match x with injd(xA) ↦ injd(xA) | injd(xB) ↦ injg(xB)`

avec le type $(A + B) \rightarrow (B + A)$

⟨2⟩ dans l'environnement $h : (x, A + B)$, typons le programme

`match x with injd(xA) ↦ injd(xA) | injd(xB) ↦ injg(xB)` avec le type $B + A$

⟨3⟩ typons le programme `x` avec le type $A + B$

⟨3⟩ CQFD (Var avec h)

⟨4⟩ dans l'environnement $h_A : (x_A, A)$, typons le programme `injd(xA)` avec le type $B + A$

⟨5⟩ typons le programme `xA` avec le type A

⟨5⟩ CQFD (Var avec h_A)

⟨4⟩ CQFD (Injd)

⟨5⟩ dans l'environnement $h_B : (x_B, B)$, typons le programme `injg(xB)` avec le type $B + A$

⟨6⟩ typons le programme `xB` avec le type B

⟨6⟩ CQFD (Var avec h_B)

⟨5⟩ CQFD (Injg)

⟨2⟩ CQFD (Match)

⟨1⟩ CQFD (Fun)

Exemple : $(A \vee B) \Rightarrow (B \vee A)$

OCaml

```
# type ('x,'y) x_plus_y = Inj_g of 'x | Inj_d of 'y;;

# Inj_g(3);;
- : (int, 'a) x_plus_y = Inj_g 3

# Inj_d(3);;
- : ('a, int) x_plus_y = Inj_d 3

# function x -> match x with
    Inj_g(xA) -> Inj_d(xA)
  | Inj_d(xB) -> Inj_g(xB);;
- : ('a, 'b) x_plus_y -> ('b, 'a) x_plus_y = <fun>
```

Langage logique = Langage de types

- langage de preuve = langage de programmation (preuve = programme)
 - ▶ exécuter un programme (une preuve) ?
 - ★ simplification de la preuve (élimination des coupures)
- on enrichit le langage de programmation pour disposer de types correspondant à la négation, et aux formules avec quantificateurs
 - ▶ mécanismes d'exceptions, types dépendants, etc.
- si le typage est décidable (on dispose d'un algorithme permettant de calculer le type d'un programme) alors on dispose d'un algorithme permettant de vérifier qu'une preuve est correcte
 - ▶ assistants à la preuve (Coq, etc.)
- c'est la vérification de la preuve qui s'obtient par un calcul ... il n'existe pas d'algorithme permettant de construire automatiquement la preuve (problème indécidable)
 - ▶ à partir d'un type on ne sait pas construire automatiquement un programme de ce type