

# Récap # 3 du français HoTT

Hugo  
Salou

## THÉORIE des TYPES

### (homotopiques)

Dans ce doc, je vais prendre une approche un peu différente de celle vue en "cours", dans l'espoir que ça soit plus clair.

Une idée importante en théorie de la démonstration est la

Correspondance de Curry - Howard !

Ce n'est pas la correspondance de Galois !

Il y a une correspondance entre

Types

&

Propositions\*

Expressions / Programmes & Preuves.

I Logique propositionnelle  
et OCaml.

Rappelons les règles de la logique propositionnelle

en impliquant que le connecteur  $\rightarrow$  :

$$\frac{}{\Gamma, A, \Delta \vdash A} \text{Ax} \quad \frac{\Gamma, A \vdash B}{\Gamma \vdash A \rightarrow B} \rightarrow_I$$

$$\frac{\Gamma \vdash A \rightarrow B}{\Gamma \vdash B} \rightarrow_E \quad \frac{\Gamma \vdash A}{\Gamma \vdash A} \rightarrow_E$$

\* pas la notion de "mère proposition" de Flotz

Avec l'idée d'une logique constructive en tête, on s'imagine qu'une preuve de  $A \rightarrow B$  serait une sorte de procédure pour transformer  
en gros, une fonction

une preuve de A en une preuve de B.  
a : A      ... : B en fonction de a.

D'où une preuve de  $A \rightarrow B$  est une famction de type  $A \rightarrow B$ .

On peut réécrire les règles logiques en règles de typage :

$$\frac{}{\Gamma, x:A, \Delta \vdash x:A} \text{Var} \quad \frac{\Gamma \models x:A \vdash e:B}{\Gamma \vdash \text{fun } x \rightarrow e : A \rightarrow B} \text{Abs}$$
$$\frac{\Gamma \vdash f : A \rightarrow B \quad \Gamma \vdash e : A}{\Gamma \vdash f(e) : B} \text{App}$$

dictionary  
(variable, type)

Ce sont les mêmes règles!

Continuons avec "\*" et "+"...

On définit

type ('a, 'b) sum =

| left of 'a

| Right of 'b

où je mettrais plutôt  $t + t'$  au lieu  
de  $(t, t')$  sum.

Les règles logiques impliquant " $\wedge$ " (et) sont:

$$\frac{\Gamma \vdash A \wedge B}{\Gamma \vdash A} \wedge_E^L \quad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash B} \wedge_E^R \quad \frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \wedge B}$$

Ces règles sont la version "logique"

des éléments :

$$\text{fst} = \text{fun } (x, y) \rightarrow x : A * B \rightarrow A$$

$$\text{snd} = \text{fun } (x, y) \rightarrow y : A * B \rightarrow B$$

$$\text{pair} = \text{fun } x \ y \rightarrow (x, y) : A \rightarrow B \rightarrow A * B$$

avec l'idée que l'on identifie

$$A * B \text{ et } A \wedge B.$$

C'est cohérent vu qu'une preuve de  $A \wedge B$

consiste en une preuve de  $A$  et une preuve

de  $B$ ... ensemble... une paire  $(a, b)$ .

De même,  $A + B$  et  $A \vee B$  se comportent  
de la même manière; et cela permet d'imaginer la mystérieuse règle  $\vee E$ :

côté logique:

$$\frac{\Gamma \vdash A}{\Gamma \vdash A \vee B}$$

côté typage:

$$\text{fun } x \rightarrow \text{left } x : A \rightarrow (A + B)$$

$$\frac{\Gamma \vdash B}{\Gamma \vdash A \vee B}$$

$\text{fun } x \rightarrow \text{Right } x :$   
 $B \rightarrow (A + B)$

$$\frac{\Gamma \vdash A \vee B \quad \Gamma \vdash A \rightarrow C \quad \Gamma \vdash B \rightarrow C}{\Gamma \vdash C}$$

→ Décurryfié

$(\text{fun } x \ f \ g \rightarrow$   
 $\text{match } x \text{ with}$   
 $| \text{Left } a \rightarrow f \ a$   
 $| \text{Right } b \rightarrow g \ b) :$

$$(A + B) \rightarrow (A \rightarrow C) \rightarrow (B \rightarrow C) \rightarrow C.$$

Par le côté générique du typage, on a :

- la capacité d'appliquer nos règles et preuves aux types que l'on veut
- l'impossibilité (ou presque) de "tricher":  
 on ne peut pas montrer  $A \rightarrow B$  en toute généralité...  
 (on oublie Gbjs. magic, lit rec  $f x = f z$  et autres)

Il y a aussi un côté "calcul" à la correspondance de Curry-Howard.

Nous verrons ça en Projet Fonctionnel !

## II Typage dépendant en Rocq (ou Coq, c'est pareil...)

Pour réaliser un  $\forall x, P(x)$  avec Curry-Howard comme un type, on doit s'autoriser un type dépendant.

Une fonction dépendante est une fonction  $f$  dont le type de retour  $f(x) : B(x)$  dépend de l'entrée  $x : A$ . On note alors

$$f : \prod_{x:A} B(x) \text{ ou } f : (x:A) \rightarrow B(x)$$

Exemple :

$$f : \prod_{x:\mathbb{Z}} \left\{ \begin{array}{l} \mathbb{N} \text{ si } x = \text{tt} \\ \mathbb{Q} \text{ si } x = \text{ff} \end{array} \right.$$

$$f(\text{tt}) := 42$$

$$f(\text{ff}) := \text{ff}$$

où  $\mathbb{2}$  est le type à deux éléments, les booléens.

Si  $B$  ne dépend pas de  $x$ , alors :

$$\prod_{x:A} B(x) = A \multimap B$$

Et, d'un point de vue logique, on encodé

le type des preuves d'un " $A$ " : c'est une

fonction prenant  $x$  et renvoyant une

preuve de  $B(x)$ .

(Avec Inductive, les  $\Pi$ -types et match, on peut reconstruire toute la logique du 1<sup>er</sup>, 2<sup>nd</sup>, ... ordre.)

Détail technique Pour éviter des paradoxes

"à la Russel", on n'a pas de type de tous

les types (paradoxe du Girard) mais

une hiérarchie infinie d'univers (le type

d'un type est un univers):

$$\mathcal{U}[0] : \mathcal{U}[1] : \mathcal{U}[2] : \dots : \mathcal{U}[n] : \dots$$

Dans la suite, on s'en fiche un peu, du moment qu'on ne crée pas de paradoxes.

On note  $\mathcal{U}$  "comme si" c'était le type des

types. (En Rocaq, cette hiérarchie est implicite, sauf Prop et Set). <sup>→ Type.</sup>

# Quelques définitions utiles (modulo abus de notations)

Inductive " $A \times B$ " : Type :=

| " $\_, \_$ " :  $A \rightarrow B \rightarrow A \times B$ .



o pour noter  $x, y$

On peut aussi faire les paires dépendantes:

Inductive " $\sum_{x:A} B(x)$ " : Type :=

| " $\_, \_$ " :  $(x:A) \rightarrow B x \rightarrow \sum_{x:A} B(x)$ .

"abus" mais ça passe:

o on peut identifier

$$\sum_{x:A} B(x) = A \times B$$

$$\sum_{x:A} B(x)$$

et

$$\exists x:A, B(x)^*$$

avec Curry-Howard

Très important : l'égalité

\* on en reparle quand  
on aura vu plus de  
HOTT

Inductive " $x =_A y$ ": Prop :=

| refl :  $x =_A x$ .

Chacune de ces définitions inducitives engendre un principe inductif adapté.

Par exemple, pour les  $\Sigma$ -types, on a :

- pour toute propriété  $P : \sum_{x:A} B(x) \rightarrow U$

- si, pour tout  $a:A$ , et tout  $b : B(a)$ ,  
on a  $P(a, b)$

- alors pour tout  $x : \sum_{a:A} B(a)$ , on a  $P(x)$ .

Celui de l'égalité est moins intuitif :

- pour toute propriété  $P : \prod_{x:A} \prod_{y:A} x =_A y \rightarrow U$

- si on a  $P(x, x, \text{refl}_x)$  pour tout  $x:A$ ,
- alors on a  $P(x, y, p)$  pour tout  $p : x =_A y$   
et  $x, y : A$

### III Flott et l'égalité.

En général, on traite " $x = y$ " comme un type pas très intéressant:

- soit on montre  $x = y$

- soit on montre  $\neg(x = y)$  où  $\neg A := A \rightarrow \text{false}$

peu importe la "tête" des éléments à l'intérieur... au final, c'est tous des refl (perfos with extra steps, genre  $\text{add}(0, n) = \underline{\mathbb{N}}^n$ ).

Mais pas en Flott!

On considère que " $x =_A y$ " est de la donnée !

By the way! Il y a deux égalités  
en théorie des types.

1) L'égalité par définition

$f \equiv g$  si  $f$  et  $g$  sont exactement  
identiques  
 $\uparrow$   
égal par def° (c.f l'annexe sur  
l' $\alpha, \beta, \gamma$ -équivalence)

2) Le type égalité

$x =_A y$

Ces deux égalités sont dans des contextes  
différents un est un type, l'autre est un  
"jugement", un "par définition, ... = ...".

On peut "passer" de  $x \equiv y$  à  $x = y$  par  
refé.

En effet, si  $x \equiv y$  alors on peut inverser  $x$  et  $y$  à tout moment.

Exemple: Si on pose:

let rec add n = function

| 0 → m

| s m → S(add n m)

alors il existe  $u, v$  tel que

$$u : \text{add}(n, 0) =_{\text{IN}} n$$

$$v : \text{add}(0, n) =_{\text{IN}} n,$$

mais

$$\text{add}(n, 0) \equiv n \text{ et } \text{add}(0, n) \not\equiv n$$



c'est vrai pour  
"on le montre"  
pas juste par  
définition

Je préfère revenir plus tard sur l'inter-  
prétation des types comme espace topologique  
en  $\mathcal{C}\text{-FOTT}$ .

# ANNEXE

$\alpha$ ,  $\beta$  et  $\eta$ -conversions

&

catégories aussi

## I. L' $\alpha$ -conversion

Gm s'autorise à renommer les variables

liées :

$$(\text{fun } x \rightarrow f(x)) \equiv_{\alpha} (\text{fun } y \rightarrow f(y))$$

$$(\text{fun } f \rightarrow^{\frac{f}{x}} f(f)) \not\equiv_{\alpha} (\text{fun } y \rightarrow g(y))$$

pas de collisions!

pas les variables libres!

Truc pas mal: indices de De Bruijn

Pareil :

$$\left( \begin{array}{l} \text{match } x \text{ with} \\ | \text{Left } u \rightarrow \dots \\ | \text{Right } v \rightarrow \dots \end{array} \right) \equiv_\Delta \left( \begin{array}{l} \text{match } x \text{ with} \\ | \text{Left } a \rightarrow \dots \\ | \text{Right } b \rightarrow \dots \end{array} \right)$$

## II La $\beta$ -réduction : en fait du calcul.

La  $\beta$ -réduction représente le fait de faire un calcul.

Par exemple

$$(\text{fun } x \rightarrow (x, x)) \ 3 \xrightarrow{\beta} (3, 3).$$

Ce n'est pas une relation d'équivalence, mais on peut considérer la  $\beta$ -équivalence comme la plus petite relation d'équivalence contenant  $\rightarrow_\beta$ .

(Nous aurons plus de détails en Pro.Fam.)

La  $\beta$ -réduction s'occupe aussi des match,  
et de la récursivité :

match Left a with

| Left x  $\rightarrow$  f(x)

| Right y  $\rightarrow$  g(y)

↓  
 $\beta$   
f(a)

De même, avec les paires,

$\text{fst}(a, b) \xrightarrow{\beta} a$

L'idée est que les éliminations éliminent  
les introductions.

### III L' $\eta$ -expansion : c'est la même chose

On peut se dire que

$(\lambda x \rightarrow f x)$  et  $f$

se comportent identiquement mais ne sont pas  $\alpha\beta$ -équivalents...

De même avec

$(\text{fst } p, \text{snd } p)$  et  $p$

et avec

`if b then true else false` et `b`

et avec

`Match x with`

`et x.`

`| Left a -> Left a`

`| Right b -> Right b`

C'est ça l' $\eta$ -expansion !

Une autre manière d'écrire l' $\eta$ -équivalence pour les paires est :

$$\frac{P =_{\eta} \text{fst } M \quad Q =_{\eta} \text{snd } M}{\langle P, Q \rangle =_{\eta} M}$$

On peut le voir comme "une paire  $M$  est définie de manière unique par  $\text{fst } M$  et  $\text{snd } M$ "

On reviendra sur l' $\alpha\beta\eta$ -équivalence la fois prochaine... avec des catégories...