Récap #3 du traquenard 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 GCAML.

Rappelons les règles de la logique propositionnelle n'impliquant que le connecteur $\to$:

$$\frac{}{\Gamma, A, \Delta \vdash A}\ Ax \qquad \frac{\Gamma, A \vdash B}{\Gamma \vdash A \to B}\ \to_I$$

$$\frac{\Gamma \vdash A \to B \qquad \Gamma \vdash A}{\Gamma \vdash B}\ \to_E$$

* pas la notion de "mere proposition" de HoTT

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

En gros, une fonction

$a : A$ $\quad\quad$ $\ldots : B$ en fonction de a.

D'où une preuve de $A \to B$ est une fonction de type $A \to B$.

On peut réécrire les règles logiques en règles de typage :

$$\frac{}{\Gamma, x : A, \Delta \vdash x : A}\ \text{Var} \qquad \frac{\Gamma\ x : A \vdash e : B}{\Gamma \vdash \text{fun } x \to e : A \to B}\ \text{Abs}$$

$$\frac{\Gamma \vdash f : A \to B \qquad \Gamma \vdash e : A}{\Gamma \vdash f(e) : B}\ \text{App}$$

dictionnaire (variable, type)

Ce sont les mêmes règles!

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

On définit

```
type ('a, 'b) sum =
  | Left of 'a
  | Right of 'b
```

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

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

$$\frac{\Gamma \vdash A \wedge B}{\Gamma \vdash A}\wedge_E^g \qquad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash B}\wedge_E^d \qquad \frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \wedge B}$$

Ces règles sont la version "logique"

des éléments :

$fst = fun\ (x,y) \to x \ : A * B \to A$

$snd = fun\ (x,y) \to y \ : A * B \to B$

$pair = fun\ x\ y \to (x,y) : A \to B \to A * B$

avec l'idée que l'on identifie

$A * B$ 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 $\vee_E$:

coté logique:

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

coté typage:

$fun\ x \to Left\ x :$
$A \to (A + B)$

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

```
fun x -> Right x :
B -> (A + B)
```

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

⇝ Décurryfié

```
(fun x f g ->
   match x with
   | Left a -> f a
   | Right b -> g b) :
(A + B) -> (A -> C) -> (B -> C) -> 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 $A \to B$ en toute généralité...
  (on oublie Obj.magic, let rec f x = f x et autres)

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

Vous verrez 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 typage <u>dépendant</u>.

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

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

<u>Exemple :</u>

$$f : \prod_{x:\mathbb{2}} \begin{cases} \mathbb{N} & \text{si } x = tt \\ \mathbb{2} & \text{si } x = ff \end{cases}$$

$$f\ (tt) := 42$$

$$f\ (ff) := ff$$

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

Si B ne dépend pas de x, alors :

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

Et, d'un point de vue logique, on encode le type des preuves d'un "$\forall$" : c'est une fonction prenant x et renvoyant une preuve de $B(x)$.

Avec Inductive, les Π-types et match, on peut reconstruire toute la logique du 1er, 2nd, ... ordre.

Détail technique Pour éviter des paradoxes "à la Russel," on n'a pas de type de tous les types (paradoxe de Girard) mais une hiérarchie infinie d'univers (le type d'un type est un univers):

$$U[0] : U[1] : U[2] : \cdots : U[n] : \cdots$$

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

On note U "comme si" c'était le type des types. (En Rocq, cette hiérarchie est implicite, sauf Prop et Set). ⇝ Type.

Quelques définitions utiles (modulo abus de notations)

```
Inductive "A × B" : Type :=
| "_,_" : A → B → A × B.
```

"_,_" → pour noter x, y

On peut aussi faire les <u>paires dépendantes</u> :

```
Inductive "Σ x:A B(x)" : Type :=
| "_,_" : (x:A) → B x → Σ x:A B(x).
```

"_,_" : "abus" mais ça passe :

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

↳ on peut identifier

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

et

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

avec Curry-Howard

<u>Très important</u> : 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 inductives engendre un principe inductif adapté.

Par exemple, pour les Σ-types, on a :

- pour toute propriété $P : \sum_{x:A} B(x) \to \mathcal{U}$
- si, pour tout $a : A$, et tout $b : B(x)$, 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 $P : \prod_{x:A} \prod_{y:A} x =_A y \to \mathcal{U}$
- si on a $P(x, x, \mathrm{refl}_x)$ pour tout $x : A$,
- alors on a $P(x, y, p)$ pour tout $p : x =_A y$ et $x, y : A$

III Hott 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 $\neg A := A \to \text{False}$

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

Mais pas en Hott!

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

$f \equiv g$ ssi f et g sont exactement identiques (c.f l'annexe sur l'α, β, η-équivalence)

↑ égal par déf°

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 definition, ... = ...".

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

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 -> n
  | S m -> S(add n m)
```

alors il existe u, v tel que

$$u : add(n, 0) =_{\mathbb{N}} n$$
$$v : add(0, n) =_{\mathbb{N}} n,$$

mais

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

C'est vrai par "on le montre" pas juste par définition

Je préfère revenir plus tard sur l'interprétation des types comme espace topologique en HoTT.

ANNEXE

α, β et η-conversions

&

catégories aussi

I. L'α-conversion

On s'autorise à renommer les variables liées :

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

$$(\underbrace{\text{fun } f \xrightarrow{\#\#_2} f(f)}) \not\equiv_\alpha (\underbrace{\text{fun } y \xrightarrow{\#\#_2} g(y)})$$

pas de collisions ! pas les variables libres !

Truc pas mal : indices de De Bruijn

Pareil :

$$\begin{pmatrix}\texttt{match } x \texttt{ with}\\ \texttt{| Left } u \texttt{ -> } \dots\\ \texttt{| Right } v \texttt{ -> } \dots\end{pmatrix} \equiv_\alpha \begin{pmatrix}\texttt{match } x \texttt{ with}\\ \texttt{| Left } a \texttt{ -> } \dots\\ \texttt{| Right } b \texttt{ -> } \dots\end{pmatrix}$$

II La β-réduction : on fait du calcul.

La β-réduction représente le fait de faire un calcul.

Par exemple

$$(\texttt{fun } x \texttt{ -> } (x,x))\ 3 \longrightarrow_\beta (3,3).$$

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

(Nous aurez plus de détails en Pro.Fon.)

La β-réduction s'occupe aussi des match, et de la récursivité :

```
match Left a with
| Left x → f(x)
| Right y → g(y)
```

$\downarrow_\beta$

$f(a)$

De même, avec les paires,

$$\mathrm{fst}\ (a, b) \longrightarrow_\beta a$$

L'idée est que les éliminations éliminent les introductions.

III L'η-expension : c'est la même chose

On peut se dire que

(fun x -> f x) et f

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

De même avec

(fst p, snd p) et p

et avec

if b then true else false et b

et avec

```
match x with
| Left a -> Left a
| Right b -> Right b
```

et x.

C'est ça l'η-expension !

Une autre manière d'écrire l'η-équivalence pour les paires est :

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

On peut le voir comme " une paire M est définie de manière unique par fst M et snd M !

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