Programmer = démontrer?
La correspondance de Curry-Howard aujourd'hui

Dixième cours

Qu'est-ce que l'égalité?
De Leibniz à la théorie homotopique des types

Xavier Leroy

Collège de France

2019-01-23

COLLÈGE
DE FRANCE
1530

Que voulons-nous dire ?

En informatique, lorsque nous écrivons dans un programme

```
e1 = e2        e1 == e2        e1 === e2        e1.equals(e2)
```

En mathématiques, lorsque nous écrivons

$$\Delta = b^2 - 4ac \qquad \frac{2}{4} = \frac{1}{2} \qquad e^{i\pi} = -1$$

En philosophie, lorsque nous parlons de l'identité des objets.
(Le bateau de Thésée est-il le même une fois changées toutes ses planches ?)

I

Quelques notions d'égalité

L'égalité dite de Leibniz

Deux objets sont égaux
si et seulement si
toute propriété vraie de l'un est aussi vraie de l'autre

En logique d'ordre supérieur, ce principe fournit une définition de l'égalité connue sous le nom «d'égalité de Leibniz» :

$$x = y \quad \stackrel{def}{=} \quad \forall P,\ P(x) \Leftrightarrow P(y)$$

Rendons à Leibniz …

Principe d'indiscernabilité des identiques : deux entités identiques possèdent les mêmes propriétés.

> *A et B sont identiques signifie qu'ils peuvent être substitués l'un à l'autre dans toutes les propriétés* salva veritate.
>
> *G. W. Leibniz,* Échantillon de calcul universel

Principe d'identité des indiscernables : si deux entités possèdent les mêmes propriétés, alors elles sont identiques.

> *[I]l n'est pas vrai que deux substances se ressemblent entièrement et soient différentes* solo numero.
>
> *G. W. Leibniz,* Discours de métaphysique

Variations sur l'égalité de Leibniz

Une axiomatisation en logique du premier ordre :

Axiome de réflexivité : $\forall x,\ x = x$

Schéma d'axiome : $\forall x, y,\ x = y \Rightarrow (P(x) \Leftrightarrow P(y))$

(un axiome par prédicat P)

De ces axiomes s'ensuit la réciproque :
si $P(x) \Leftrightarrow P(y)$ pour tout prédicat P,
on prend $\lambda z.\ (x = z)$ pour P et on a $x = x \Leftrightarrow x = y$,
d'où $x = y$.

Variations sur l'égalité de Leibniz

On peut remplacer l'équivalence $P(x) \Leftrightarrow P(y)$ par une implication :

$$x = y \quad \stackrel{def}{=} \quad \forall P,\ P(x) \Rightarrow P(y)$$

Malgré l'apparente dissymétrie, la définition est équivalente : car si $\forall P,\ P(x) \Rightarrow P(y)$, en prenant $P = \lambda z.\ P(z) \Rightarrow P(x)$, il vient

$$(P(x) \Rightarrow P(x)) \Rightarrow (P(y) \Rightarrow P(x)) \qquad \text{et donc} \qquad P(y) \Rightarrow P(x)$$

D'où $\forall P,\ P(y) \Rightarrow P(x)$.

Relations d'équivalence

Une relation R est une relation d'équivalence si elle est

$$\begin{array}{ll} \text{réflexive :} & \forall x,\ R(x,x) \\ \text{symétrique :} & \forall x, y,\ R(x,y) \Rightarrow R(y,x) \\ \text{transitive :} & \forall x, y, z,\ R(x,y) \wedge R(y,z) \Rightarrow R(x,z) \end{array}$$

Une autre définition de l'égalité sur un ensemble A : c'est la plus fine des relations d'équivalence sur A, c.à.d. l'intersection de toutes ces relations

$$\begin{array}{rcl} x =_A y & \stackrel{def}{=} & (x,y) \in \bigcap \{R \mid R \text{ relation d'équivalence sur } A\} \\ & \stackrel{def}{=} & R(x,y) \text{ pour toute relation d'équivalence } R \text{ sur } A \end{array}$$

Cette définition est équivalente à l'égalité de Leibniz. (Exercice.)

L'égalité en théorie des types

Per Martin-Löf introduit dans la version 1973 de sa théorie des types un type $x =_A y$ des identités entre $x, y : A$.

Un élément du type $x =_A y$ est une preuve de l'égalité de x et y.

Ce type a un constructeur $\mathtt{refl}_A$ et un éliminateur J_A.

$$\mathtt{refl}_A : \forall x : A.\ x =_A x$$

$$J_A : \forall C : (\forall x, y : A.\ x =_A y \to Set).\ (\forall z : A.\ C\ z\ z\ (\mathtt{refl}_A\ z)) \to (\forall x, y : A.\ \forall s : x =_A y.\ C\ x\ y\ s)$$

tels que $J_A\ C\ d\ a\ a\ (\mathtt{refl}_A(a)) = d\ a : C\ a\ a\ (\mathtt{refl}_A(a))$.

L'égalité en théorie des types

$$J_A : \forall C : (\forall x, y : A.\ x =_A y \to Set).$$
$$(\forall z : A.\ C\ z\ z\ (\mathtt{refl}_A\ z)) \to (\forall x, y : A.\ \forall s : x =_A y.\ C\ x\ y\ s)$$

En bon français : soit C un prédicat à 3 arguments, x et y de type A et $s : x =_A y$ une preuve de l'égalité entre x et y.

Pour que C soit toujours vrai, il suffit qu'il soit vrai dans le cas où x et y sont la même variable z et s est l'égalité triviale $\mathtt{refl}_A\ z$.

Exemple : l'indiscernabilité des identiques !

Soit $P : A \to Set$ un prédicat. On prend $C\ x\ y\ s = (P\ x \to P\ y)$. Clairement $C\ z\ z\ (\mathtt{refl}_A\ z) = P\ z \to P\ z$ est vrai. Donc, pour toute preuve de $x =_A y$, on a $P\ x \to P\ y$.

L'égalité en théorie des types

Le type $x =_A y$, interprété via Curry-Howard comme une proposition, est équivalent à l'égalité de Leibniz :

- Réflexivité : le type $x =_A x$ est habité par $\mathtt{refl}_A\ x$.
- Indiscernabilité des identiques : si $x =_A y$ est habité, alors $P\ x \to P\ y$ pour tout prédicat $P : A \to \mathit{Set}$.

De plus, on peut ≪calculer sur les preuves d'égalité≫ de manière effective. Par exemple, avec J on peut définir des termes

$$\mathtt{sym}_A : \forall x, y : A.\ x =_A y \to y =_A x$$

$$\mathtt{trans}_A : \forall x, y, z : A.\ x =_A y \to y =_A z \to x =_A z$$

qui vérifient $\mathtt{trans}_A\ x\ y\ x\ s\ (\mathtt{sym}_A\ x\ y\ s) \xrightarrow{*} \mathtt{refl}_A\ x$.

L'égalité comme prédicat inductif

Si on dispose de familles inductives, c.à.d. de prédicats inductifs, comme en Agda et en Coq, on peut définir l'égalité comme un prédicat inductif.

```
Inductive eq (A: Type): A -> A -> Prop :=
  | eq_refl: forall (x: A), eq A x x.
```

Coq utilise la variante ci-dessous, logiquement équivalente mais parfois plus pratique d'emploi :

```
Inductive eq (A: Type) (x: A): A -> Prop :=
  | eq_refl: eq A x x.
```

L'égalité comme prédicat inductif

```
Inductive eq (A: Type) (x: A): A -> Prop :=
  | eq_refl: eq A x x.
```

Le principe d'induction pour ce prédicat inductif, engendré automatiquement par Coq, est le principe d'indiscernabilité des identiques :

```
eq_ind: forall (A : Type) (x : A) (P : A -> Prop),
        P x -> forall y : A, eq A x y -> P y
```

Il s'ensuit que ce prédicat `eq` est équivalent à l'égalité de Leibniz :

```
forall (A: Type) (x y: A),
eq A x y <-> forall (P: A -> Prop), P x -> P y.
```

L'égalité comme prédicat inductif

```
Inductive eq (A: Type) (x: A): A -> Prop :=
    eq_refl: eq A x x.
```

Toutes les utilisations («éliminations») d'une égalité se ramènent à des filtrages sur des termes de type `eq A x y`. Par exemple, le principe d'indiscernabilité des identiques :

```
Definition F (A: Type) (P: A -> Prop) (x y: A) (s: eq A x y)
           : P x -> P y :=
  match s with
  | eq_refl _ _ => fun p => p
  end.
```

Exercice : définir l'éliminateur J_A de Martin-Löf sous forme d'un filtrage.

II

Grandeur et misère de l'égalité en Coq

Égalité sur les types inductifs simples

L'égalité de la théorie des types se comporte très bien sur les types de données purement inductifs comme `nat` ou `nat * bool` ou `list nat`.

En particulier, on a la propriété d'extensionnalité : deux structures de données sont égales si et seulement si leurs composantes sont égales. Par exemple, pour deux listes :

$$[x_1; \ldots; x_p] = [y_1; \ldots; y_q] \quad \text{ssi} \quad p = q \text{ et } x_i = y_i \text{ pour i = 1, } \ldots p$$

De plus, l'égalité est décidable : pour un type A purement inductif, on peut définir une fonction `beq` : $A \to A \to$ `bool` telle que `beq` x y renvoie `true` ssi $x = y$ et renvoie `false` ssi $x \neq y$.

```
Fixpoint beq_nat (p q: nat) : bool :=
  match p, q with
  | O, O => true | S p, S q => beq_nat p q | _, _ => false
  end
```

Égalité entre fonctions

Deux fonctions sont égales si elles sont convertibles. Par exemple, avec la définition de Coq pour + :

$$\texttt{(fun x => 1 + x)} \quad =_\beta \quad \texttt{(fun x => S x)} \quad =_\eta \quad \texttt{S}$$
$$\texttt{(fun x => x + 1)} \quad \neq_{\beta\eta} \quad \texttt{S}$$

C'est le seul moyen de démontrer que deux fonctions sont égales. En particulier on ne peut pas démontrer de principe d'extensionnalité (deux fonctions qui ont le même graphe sont égales).

FE (*Function Extensionality*)

$$\forall A, B : \texttt{Type}.\ \forall f, g : A \to B.\ (\forall x : A,\ f\ x = g\ x) \to f = g$$

DFE (*Dependent Function Extensionality*)

$$\forall A : \texttt{Type}.\ \forall B : A \to \texttt{Type}.\ \forall f, g : \Pi(x : A).\ B\ x.\ (\forall x : A,\ f\ x = g\ x) \to f = g$$

Égalité entre fonctions

Supposons deux fonctions $f, g : A \to B$ telles que $\forall x : A.\ f\ x = g\ x$.

Un argument à base de relations logiques peut montrer l'équivalence contextuelle entre f et g, d'où il s'ensuit que $P\ f$ et $P\ g$ sont logiquement équivalents pour tout $P : (A \to B) \to \texttt{Prop}$.

Mais c'est un argument ≪méta≫ qu'on ne peut pas montrer à l'intérieur de la théorie des types !

En fait, **FE** et son extension **DFE** sont indépendantes de CC + univers :

- Les modèles ensemblistes valident **DFE**
- Un modèle syntaxique de Boulier, Pédrot, Tabareau (2017) l'invalide.

Égalité entre types coinductifs

Tout comme pour les fonctions, l'égalité sur les types coinductifs n'est pas extensionnelle : deux flux s_1, s_2 tels que

$$\texttt{hd}(\texttt{tl}^n(s_1)) = \texttt{hd}(\texttt{tl}^n(s_2)) \quad \text{pour tout } n$$

ne vérifient pas $s_1 = s_2$ en général.

L'usage est de raisonner non pas sur l'égalité entre flux, mais sur leur bisimilarité, notion définie par un prédicat coinductif :

```
CoInductive bisim (A: Type): stream A -> stream A -> Prop :=
| bisim_intro: forall s1 s2,
    hd s1 = hd s2 -> bisim (tl s1) (tl s2) -> bisim s1 s2.
```

L'axiome d'extensionalité des flux $\forall s_1, s_2.\ \texttt{bisim}\ s_1\ s_2 \Rightarrow s_1 = s_2$
est vraisemblablement indépendant de la logique de Coq.

Égalité entre termes de preuve

Une valeur d'un type (co-)inductif peut contenir des termes de preuve : des valeurs d'un type P : `Prop` qui représente une proposition logique.

Exemple : le type sous-ensemble $\{x : A \mid P(x)\}$ défini comme

```
Inductive sig (A: Type) (P: A -> Prop) : Type :=
  | exist: forall (x: A), P x -> sig A P.
```

Une valeur du type $\{x : A \mid P(x)\}$ est une paire d'un $x : A$ et d'une preuve de $P\ x$.

Égalité entre termes de preuve

Pour montrer que deux valeurs du type $\{x : A \mid P(x)\}$ sont égales, il faut non seulement montrer que leurs premières composantes x sont égales, mais aussi que leurs deux preuves de $P\ x$ sont égales. Cette notion d'égalité entre termes de preuve réserve bien des surprises !

Exemple : on cherche à définir $\mathbb{Z}$ comme un quotient de $\mathbb{N} \times \mathbb{N}$.

```
Definition Z := { p: nat * nat | fst p = 0 ∨ snd p = 0 }
```

Il y a deux preuves de $0 = 0 \vee 0 = 0$: celle qui dit que la partie gauche est vraie, et celle qui dit que la partie droite est vraie. D'où deux définitions pour le zéro de `Z` :

```
Definition zero  : Z := exist _ (0,0) (or_introl eq_refl).
Definition zero' : Z := exist _ (0,0) (or_intror eq_refl).
```

On ne peut pas montrer que `zero` $=$ `zero'`.
(Ni que `zero` $\neq$ `zero'`, d'ailleurs.)

Unicité des termes de preuve

On voudrait que deux valeurs du type $\{x : A \mid P(x)\}$ soient égales dès que leurs leurs premières composantes x sont égales.

Trois possibilités :

1. Montrer que les preuves de $P(x)$ sont uniques :
   pour tous $p, q : P(x)$ on a $p = q$.
   Souvent non démontrable ; toujours difficile.

2. Changer P en un prédicat Q équivalent et qui a la propriété d'unicité des preuves, typiquement une égalité Booléenne $f\ x = \texttt{true}$.

3. Prendre comme axiome l'insignifiance des démonstrations :

   PI (*Proof Irrelevance*) $\quad \forall P : \texttt{Prop}.\ \forall p, q : P.\ p = q$

Unicité des preuves d'égalité

Un cas particulier important est l'unicité des preuves d'égalité :

UIP(A) (*Uniqueness of Identity Proofs*) $\quad \forall x, y : A.\ \forall p, q : x = y.\ p = q$

On sait montrer cette propriété pour un certain nombre de types A, en particulier ceux où l'égalité est décidable : $\forall x, y : A.\ x = y \vee x \neq y$.
Ceci inclut les types purement inductifs comme `bool` et `nat`.

(D'où l'idée de remplacer $\{x \mid P\ x\}$ par $\{x \mid f\ x = \texttt{true}\}$.)

On peut aussi prendre UIP comme axiome, pour un type donné, ou pour tous les types.

Point d'étape

L'égalité à la manière de la théorie des types est parfaite pour les types purement inductifs, mais ne permet pas d'identifier des choses que l'on voudrait bien voir égales :

- deux fonctions qui ont le même graphe ;
- deux flux qui sont bisimilaires ;
- deux preuves de la même propriété ;
- deux propositions logiques P, Q qui sont équivalentes $P \Leftrightarrow Q$.

Approche 1 : les quasi-ensembles (*setoids*)

On peut travailler systématiquement sur des types A munis chacun d'une relation d'équivalence eq_A qui est la notion d'égalité souhaitée, p.ex.

$$eq_{A\to B}\ f\ g = \forall x : A.\ eq_B\ (f\ x)\ (g\ x)$$

Le point pénible est qu'il faut démontrer la compatibilité de chaque définition de fonction ou de prédicat :

pour toute fonction $f : A \to B$, montrer $\forall x, y : A.\ eq_A\ x\ y \to eq_B\ (f\ x)\ (f\ y)$
pour tout prédicat $P : A \to \texttt{Prop}$, montrer $\forall x, y : A.\ eq_A\ x\ y \to P\ x \Leftrightarrow P\ y$

Coq fournit quelques notations et tactiques pour faciliter ce style.

Approche 2 : ajouter des axiomes

Les axiomes les plus courants :

$$\forall A, B : \mathtt{Type}.\ \forall f, g : A \to B.\ (\forall x : A,\ f\ x = g\ x) \to f = g \qquad \textbf{(FE)}$$

$$\forall A : \mathtt{Type}.\ \forall B : A \to \mathtt{Type}.\ \forall f, g : \Pi(x : A).\ B\ x.\ (\forall x : A,\ f\ x = g\ x) \to f = g \qquad \textbf{(DFE)}$$

$$\forall P, Q : \mathtt{Prop},\ (P \Leftrightarrow Q) \Rightarrow P = Q \qquad \textbf{(PE)}$$

$$\forall P : \mathtt{Prop}.\ \forall p, q : P.\ p = q \qquad \textbf{(PI)}$$

$$\forall x, y : A.\ \forall p, q : x = y.\ p = q \qquad \textbf{(UIP)}$$

On a des raisons de croire que ces axiomes sont cohérents avec CC + univers. Avec Coq tout entier c'est moins clair.

Aussi : **PE** et **PI** reposent sur le statut très particulier de `Prop` en Coq, et n'ont pas de sens dans d'autres théories des types (Agda p.ex.).

Approche 3 : penser l'égalité autrement

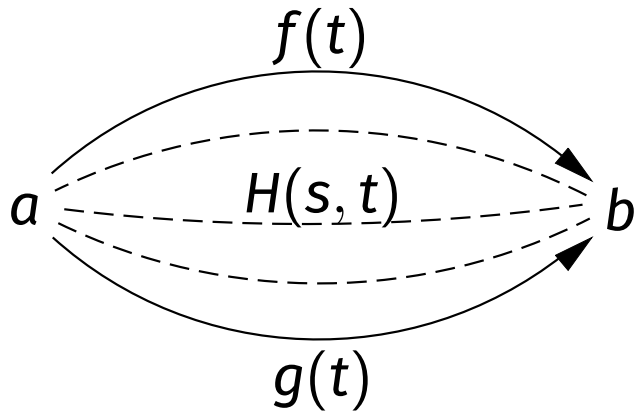
Un changement de perspective sur l'égalité pourrait nous aider à y voir plus clair…

III

Égalité et homotopie

L'homotopie

Un outil de la topologie algébrique qui s'intéresse aux déformations continues entre deux objets topologiques.

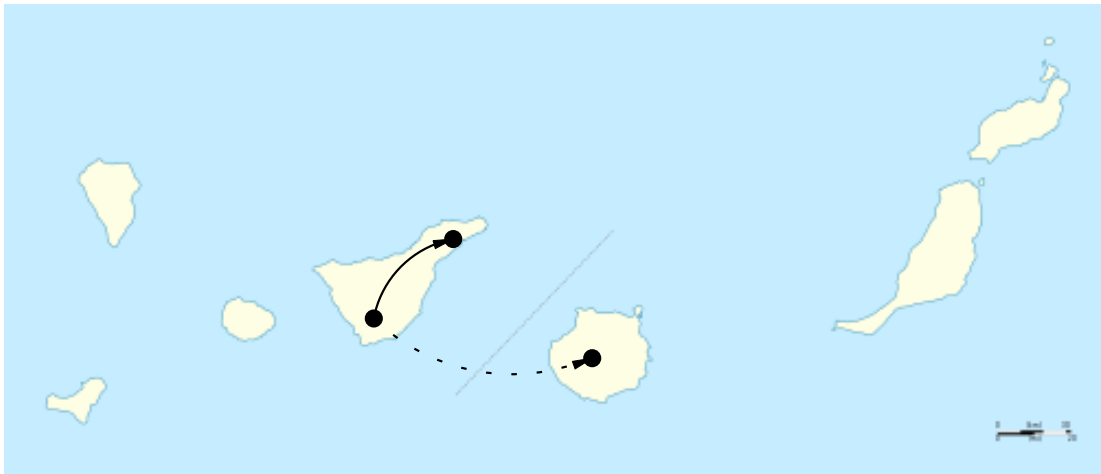


Exemple d'objet topologique :
un chemin entre deux points a, b de l'espace A est une fonction continue $f : [0,1] \to A$ telle que $f(0) = a$ et $f(1) = b$.

Exemple de déformation continue :
deux chemins f, g de a vers b sont homotopes s'il existe une fonction continue $H : [0,1] \times [0,1] \to A$ telle que $H(0,t) = f(t)$ et $H(1,t) = g(t)$ et $H(s,0) = a$ et $H(s,1) = b$.

Liberté, égalité, connexité

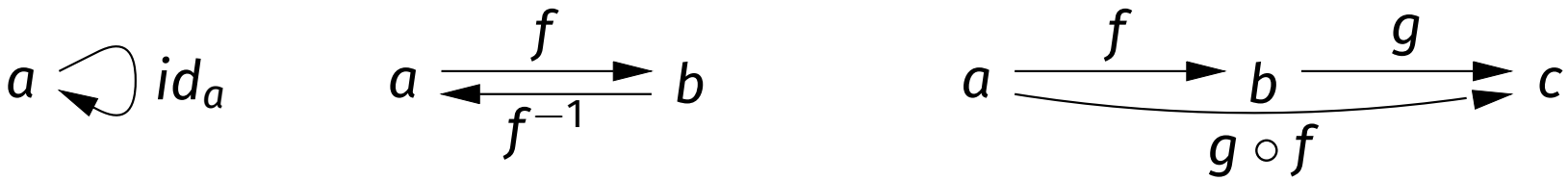
Dans l'archipel *A*, la tradition veut que deux habitants de *A* sont égaux s'il existe un chemin (terrestre) qui les relie.



Deux habitants d'une même île sont égaux.

Deux habitants de deux îles différentes sont différents.
(Sauf s'il existe un pont entre les îles.)

Opérations sur les chemins



Unité : pour chaque point a on a un chemin trivial id_a de a vers a

$$id_a = t \in [0,1] \mapsto a$$

Inverse : pour tout chemin f de a vers b, on a un chemin f^{-1} de b vers a

$$f^{-1} = t \in [0,1] \mapsto f(1-t)$$

Composition : pour tous chemins f de a vers b et g de b vers c, on a un chemin $g \circ f$ de a vers c

$$g \circ f = \begin{cases} t \in [0, \frac{1}{2}] \mapsto f(2t) \\ t \in]\frac{1}{2}, 1] \mapsto g(2t-1) \end{cases}$$

Corollaire

La relation «être relié par un chemin» est une relation d'équivalence.

Un groupoïde ?

$$
\begin{aligned}
id &: Ch(a,a) \\
\cdot^{-1} &: Ch(a,b) \to Ch(b,a) \\
\cdot \circ \cdot &: Ch(b,c) \to Ch(a,b) \to Ch(a,c)
\end{aligned}
$$

On aimerait reconnaître ici un groupoïde, c'est-à-dire

- un groupe où l'opération binaire $\circ$ est partielle ;
- une catégorie où chaque flèche a une flèche inverse.

Mais il faudrait que les identités suivantes soient valides :

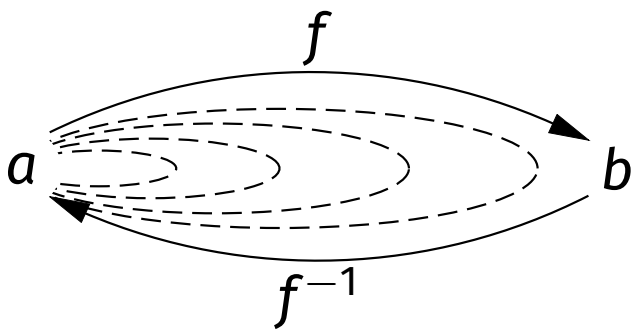
$$
\begin{aligned}
f \circ f^{-1} &= id & \qquad f^{-1} \circ f &= id \\
f \circ id &= f & \qquad id \circ f &= f \\
(f \circ g) \circ h &= f \circ (g \circ h)
\end{aligned}
$$

Égalité entre chemins = homotopie

$$f^{-1} \circ f = id_a \quad \text{pour tout } f : Ch(a, b)$$

Vue comme égalité entre fonctions, cette propriété est fausse :
id_a est une fonction constante, alors que $f^{-1} \circ f$ parcourt $a \cdots b \cdots a$.

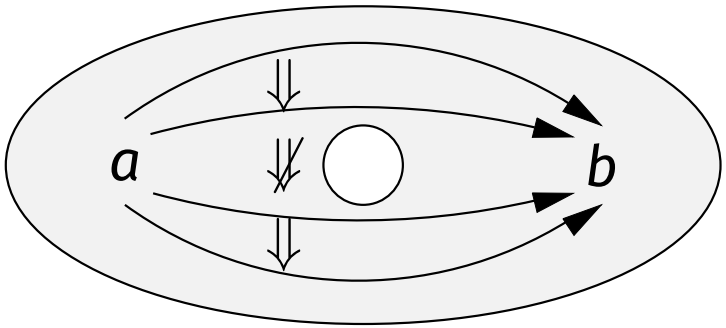
Vue comme relation d'homotopie, cette propriété est vraie :
les chemins id_a et $f^{-1} \circ f$ sont homotopes.

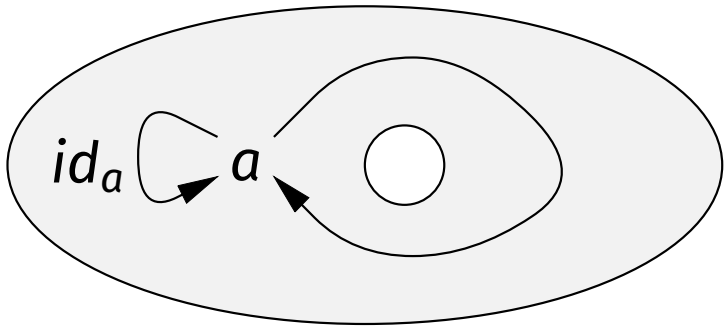


$$H(s,t) = \begin{cases} f(s \times 2t) & \text{si } t \leq \frac{1}{2} \\ f(s \times 2(1-t)) & \text{si } t > \frac{1}{2} \end{cases}$$

Chemins homotopes, chemins non homotopes

En général, deux chemins de a vers b ne sont pas homotopes, et un lacet (un chemin de a vers a) n'est pas homotope à id_a.





Cependant, les chemins intervenant dans les équations des groupoïdes sont toujours homotopes, quelle que soit la topologie de l'espace A.
En lisant $=$ comme «homotope» :

$$f \circ f^{-1} = id \qquad f^{-1} \circ f = id$$

$$f \circ id = f \qquad id \circ f = f$$

$$(f \circ g) \circ h = f \circ (g \circ h)$$

Turtles all the way down

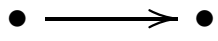
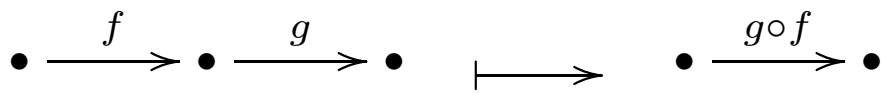
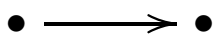
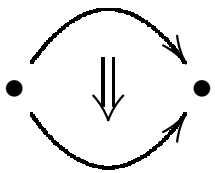
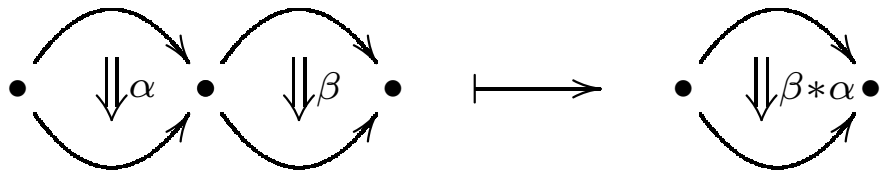
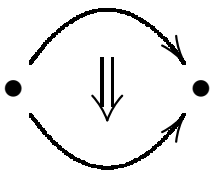
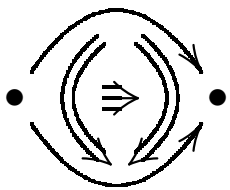
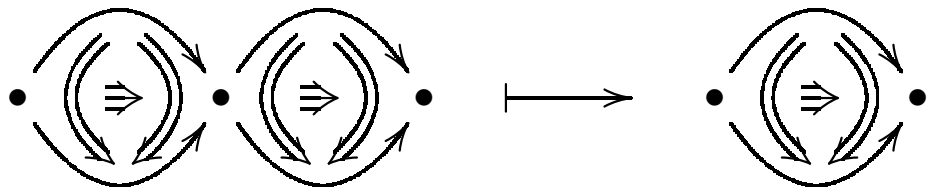
Les homotopies entre chemins forment elles-mêmes un groupoïde, avec

- des homotopies identités id_f (où f est un chemin donné);
- une loi de composition;
- un inverse.

Ces opérations satisfont les lois des groupoïdes à condition d'interpréter l'égalité entre homotopies comme l'existence d'une homotopie ≪de niveau 2≫ : une fonction continue $\Phi : [0,1]^3 \to A$ telle que $\Phi(0, \_, \_)$ égale la première homotopie et $\Phi(1, \_, \_)$ égale la seconde.

On peut répéter cette construction pour tout niveau k, obtenant ainsi un ω-groupoïde ou ∞-groupoïde.

Les trois premiers niveaux

Niveau	Objets	Identités	Compositions (transitivité)
0			
1			
2			

(Source : Cheng et Lauda, *Higher-Dimensional Categories : an illustrated guide book*.)

Théorie des types et théorie de l'homotopie

La présentation de l'égalité en théorie des types (type $x =_A y$, constructeur $\texttt{refl}_A$, éliminateur J_A) engendre naturellement un ω-groupoïde pour chaque type A.

(van den Berg et Garner, 2008 ; Lumsdaine, 2009)

Réciproquement, les ω-groupoïdes et les homotopies d'ordre supérieur fournissent des modèles de la théorie des types avec égalité.

Par exemple, Hofmann et Streicher (1998) ont utilisé des 1-groupoïdes pour construire un modèle où UIP est faux, c.à.d. où il existe deux preuves différentes d'une égalité $a = b$.

IV

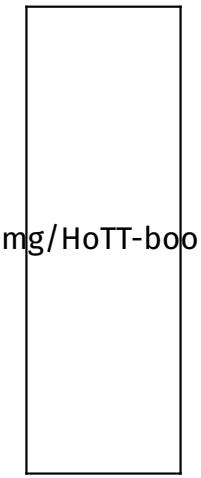
La théorie homotopique des types

La théorie homotopique des types

En très bref : c'est une théorie des types, proche de celle de Martin-Löf, mais expliquée, revue et étendue à la lumière de l'homotopie d'ordre supérieur.

Origine : une rencontre récente (2005–2010) entre un mathématicien (Voevodsky), des catégoriciens (Awodey, Warren, …) et des informaticiens logiciens (Streicher, Coquand, …).

Livre de référence : l'ouvrage collectif *Homotopy Type Theory : Univalent Foundations of Mathematics*, 2013, disponible sur le Web.



Les types de HoTT

HoTT part des mêmes types que MLTT :

$A, B ::= \mathtt{empty} \mid \mathtt{unit} \mid \mathtt{bool}$		types énumérés (0, 1, 2 éléments)
	$\mid \Pi x : A.\, B \mid \Sigma x : A.\, B$	produits dépendants, sommes dépendantes
	$\mid A + B$	sommes
	$\mid a =_A b$	identités (preuves d'égalité)
	$\mid U$	nom d'univers

Abréviations usuelles : $A \to B$ est $= \Pi\_ : A.\, B$; $A \times B$ est $\Sigma\_ : A.\, B$.

Classer les types suivant leurs égalités

On distingue deux familles importantes de types :

- Les propositions (*mere propositions* dans le livre).
  Ce sont les types A tels que toutes leurs valeurs sont égales :
  $prop(A) \stackrel{def}{=} \Pi x, y : A.\, x =_A y.$
- Les ensembles (*sets* dans le livre).
  Ce sont les types A tels que les identités sont uniques :
  $set(A) \stackrel{def}{=} \Pi x, y : A.\, \mathtt{prop}(x =_A y) \stackrel{def}{=} \Pi x, y : A.\, \Pi p, q : x =_A y.\, p = q$

En d'autres termes : les ensembles sont les types qui satisfont déjà UIP, et les propositions sont les types qui satisfont déjà PI.

Exemples de propositions

Sont des propositions :

- `unit` ($\approx$ le vrai)
- `empty` ($\approx$ l'absurde)
- $A \to B$ si B est une proposition (+ axiome FE)
- $A \to$ `empty` ($\approx$ négation) (+ axiome FE)
- $\Pi x : A.\, B(x)$ si $B(x)$ est une proposition pour tout $x : A$ (+ axiome DFE)
- $A \times B$ si A et B sont des propositions.
- $A + B$ si A et B sont des propositions et $A \to B \to$ `empty`.

$A + B$ n'est en général pas une proposition :
p.ex. `unit` $+$ `unit` a deux valeurs différentes, `inl tt` et `inr tt`.

$\Sigma x : A.\, B(x)$ n'est en général pas une proposition

Exemples d'ensembles

- Les types énumérés : `empty`, `unit`, `bool`.
- $A \to B$ si B est un ensemble (+ axiome FE)
- $\Pi x : A.\, B(x)$ si $B(x)$ est un ensemble p.t. $x : A$ (+ axiome DFE)
- $A \times B$ et $A + B$ si A et B sont des ensembles.
- $\Sigma x : A.\, B(x)$ si A et $B(x)$ p.t. $x : A$ sont des ensembles.
- A si A est une proposition.

Une correspondance propositions / types

On voudrait représenter les propositions de la logique d'ordre supérieur par des types qui soient des propositions au sens de HoTT, c.à.d. des types A tels que $x = y$ pour tous $x, y : A$.

On va utiliser un opérateur de troncature propositionnelle :
un type $\|A\|$ qui est une proposition, pour tout type A.

$$[\top] = \mathtt{unit} \qquad [\bot] = \mathtt{empty}$$
$$[P \Rightarrow Q] = [P] \to [Q] \qquad [\neg P] = [P] \to \mathtt{empty}$$
$$[P \wedge Q] = [P] \times [Q] \qquad [P \vee Q] = \|[P] + [Q]\|$$
$$[\forall x : A.\, P] = \Pi x : A.\, [P] \qquad [\exists x : A.\, P] = \|\Sigma x : A.\, [P]\|$$

La troncature propositionnelle

Deux opérations sur le type $\|A\|$:

$$\texttt{img} : A \to \|A\|$$
$$\texttt{lift} : (A \to B) \to (\|A\| \to B) \quad \text{si } B \text{ est une proposition}$$

avec $\texttt{img}\, x = \texttt{img}\, y$ p.t. $x, y : A$ et $\texttt{lift}\, f\, (\texttt{img}\, x) = f\, x$ p.t. $x : A$.

$\texttt{img}\, a$ efface toute information sur la valeur de a. Son résultat indique juste que le type A est habité.

Si $f : A \to B$ et B est une proposition, la fonction f renvoie le même résultat quel que soit son argument $a : A$. Tout ce qui lui importe est que A est habité. On peut donc la transformer en une fonction $\texttt{lift}\, f : \|A\| \to B$.

La troncature propositionnelle

$$\texttt{img} : A \to \|A\|$$
$$\texttt{lift} : (A \to B) \to (\|A\| \to B) \quad \text{si } B \text{ est une proposition}$$

Dans le cas du codage de $P \vee Q$ par $\|[P] + [Q]\|$, on cache lequel de P ou Q est vrai. On ne peut donc pas écrire de fonction $f : [P \vee Q] \to \texttt{bool}$ qui est `true` si on est dans le cas P et `false` sinon.

Mais `lift` permet quand même de faire une analyse de cas «P vrai ? Q vrai ?» afin de conclure une proposition R.

$$\frac{p : [P]}{\texttt{img}(\texttt{inl}\ p) : [P \vee Q]} \qquad \frac{q : [Q]}{\texttt{img}(\texttt{inr}\ q) : [P \vee Q]}$$

$$\frac{a : [P \vee Q] \quad f : [P] \to [R] \quad g : [Q] \to [R]}{\texttt{lift}\ (\lambda x.\ \texttt{match}\ x\ \texttt{with}\ \texttt{inl}\ p \Rightarrow f\ p \mid \texttt{inr}\ q \Rightarrow g\ q)\ a : [R]}$$

Les types inductifs d'ordre supérieur

(**HIT**, *Higher-Inductive Types*.)

Dans un type inductif usuel, les constructeurs engendrent les valeurs du type :

```
Inductive nat: Type :=
| O: nat
| S: nat -> nat.
```

Un type inductif d'ordre supérieur peut aussi avoir des constructeurs qui engendrent des chemins entre valeurs du type, c.à.d. des égalités en plus de l'égalité par défaut, et même des chemins d'ordre supérieur, c.à.d. des égalités entre égalités.

```
Inductive Z4: Type :=
| O: Z4
| S: Z4 -> Z4
| mod4: S(S(S(S O))) = O.
```

HIT = type inductif + équations

La définition de $\mathbb{Z}$ comme type inductif ≪libre≫ :

```
Inductive Z :=
  | Z0: Z
  | Zpos: positive -> Z
  | Zneg: positive -> Z.
```

Une définition ≪avec deux zéros≫ et une équation entre eux :

```
Inductive Z :=
  | Zpos: nat -> Z
  | Zneg: nat -> Z
  | Zzero: Zneg 0 = Zpos 0.
```

$\mathbb{Z}$ engendré par `0`, successeur (`S`), et son inverse le prédécesseur (`P`) :

```
Inductive Z :=
  | 0: Z    | S: Z -> Z    | P: Z -> Z
  | SP: forall z, S (P z) = z
  | PS: forall z, P (S z) = z.
```

Analyse de cas sur un HIT

```
Inductive Z4: Type :=
| O: Z4
| S: Z4 -> Z4
| mod4: S(S(S(S O))) = O.
```

La déclaration nous donne ≪gratuitement≫ une égalité, `mod4`.
Maintenant, il va falloir respecter cette égalité dans tous les calculs qui examinent une valeur de type `Z4` :

```
match (n: Z4) with O => a | S m => f m end
```

doit produire le même résultat si $n = \mathtt{O}$ et si $n = \mathtt{S}(\mathtt{S}(\mathtt{S}(\mathtt{S}\ \mathtt{O})))$,
d'où l'obligation de montrer $f\ (\mathtt{S}(\mathtt{S}(\mathtt{S}\ \mathtt{O}))) = a$.

```
Definition pred (n: Z4) :=
  match n with
  | O => S(S(S O))
  | S m => m
  end.
```

✔

```
Definition pred (n: Z4) :=
  match n with
  | O => O
  | S m => m
  end.
```

✘

Récurseurs sur un HIT

Cette obligation de preuve apparaît dans le type du récurseur (la fonction d'ordre supérieur qui fait analyse de cas + récursion).

Pour un type inductif ordinaire comme `nat`, le récurseur prend un argument par constructeur de valeur :

```
nat_rec : ∀X: Type. X → (X → X) → nat → X
```

cas `O` cas `S`

Pour un HIT comme `Z4`, le récurseur prend un argument par constructeur de valeur ou de chemin, et dans ce dernier cas c'est une preuve d'égalité :

```
Z4_rec : ∀X: Type. ∀z: X. ∀s: X → X. s(s(s(s z))) = z → Z4 → X
```

cas `O` cas `S` cas `mod4`

Récurseurs sur un HIT

```
Z4_rec : ∀X: Type. ∀z: X. ∀s: X → X. s(s(s(s z))) = z → Z4 → X
```

Armés de ce récurseur, il est facile de définir la fonction prédecesseur :

```
Definition pred : Z4 → Z4 :=
  Z4_rec Z4 (S(S(S O))) (fun m => m) (eq_refl (S(S(S O)))).
```

L'addition est tout aussi simple :
(on suit le schéma $\texttt{add}\ m\ \texttt{O} = m$ et $\texttt{add}\ m\ (\texttt{S}\ n) = \texttt{S}(\texttt{add}\ m\ n)$)

```
Definition add (m: Z4) : Z4 → Z4 :=
  Z4_rec Z4 m S (p m).
```

Le terme `p` doit prouver $\forall m,\ \texttt{S}(\texttt{S}(\texttt{S}(\texttt{S}\ m))) = m$. Cela se démontre par récurrence sur m, à l'aide d'un récurseur à type dépendant un peu plus compliqué.

(Voir l'article de Basold *et al* mentionné en bibliographie.)

La troncature comme un HIT

La troncature propositionnelle $\|A\|$ se définit par un HIT très simple :

```
Inductive tr (A: Type) : Type :=
  | img: A → tr A
  | tr_prop: ∀x y: tr A, x = y.
```

Le constructeur `tr_prop` affirme que `tr A` est une proposition.

Le récurseur associé est :

```
tr_rec (A: Type) :
   ∀X: Type. ∀i: A → X. (∀x y: X. x = y) → tr A → X
```

On voit donc qu'il ne s'applique qu'à des types `X` qui sont des propositions. Mais étant donnés un type `X` et une preuve `pX: prop X`, on définit facilement le relèvement d'une fonction `A → X` :

```
lift (f: A → X) : tr A → X := tr_rec X f pX
```

Les types quotients comme un HIT

En théorie des ensembles, le quotient A/R d'un ensemble A par une relation d'équivalence R sur A est l'ensemble des classes d'équivalence de R.

Étant donnés A : `Type` et R : $A \to A \to$ `Type`, on peut définir un type quotient A/R comme le HIT suivant :

```
Inductive Q : Type :=
  | img: A → Q
  | img_eq: forall x y, R x y → img x = img y.
```

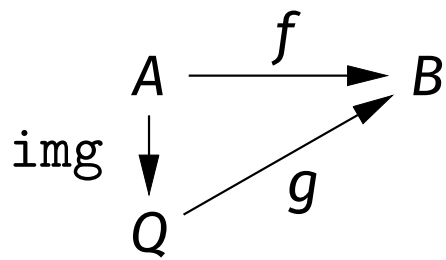
En supposant que R est une relation d'équivalence, on peut montrer la réciproque de `img_eq`, ce qui montre que

```
forall x y,  img x = img y <-> R x y.
```

Les types quotients comme un HIT

```
Inductive Q : Type :=
  | img: A → Q
  | img_eq: forall x y, R x y → img x = img y.
```

Étant donnée une fonction $f : A \to B$ qui est compatible avec R (c.à.d. $R\ x\ y \Rightarrow f\ x = f\ y$), on veut construire une fonction $g : Q \to B$:



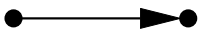
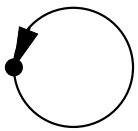
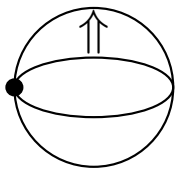
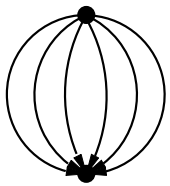
Il suffit de prendre

```
Definition g (q: Q) := match q with img a => f a end
```

ou, plus exactement, son équivalent en termes du récurseur `Q_rec`. On obtient que $g\ (\texttt{img}\ a) = f\ a$ pour tout $a : A$, comme attendu.

Les HITs pour l'homotopie

Les HITs permettent de décrire des espaces topologiques de manière purement synthétique :

Intervalle | Cercle | Sphère | Suspension

```
  0: I
  1: I
seg: 0=1
```

```
base: S1
loop: base = base
```

```
base: S2
surf: refl_base = refl_base
```

```
  N: Susp
  S: Susp
merid: A -> N = S
```

V

Pour aller plus loin

Équivalences et univalence

$f : A \to B$ est une équivalence si c'est une bijection qui ≪se comporte bien≫ vis-à-vis des chemins d'égalité :

$$\Pi y : B.\ \mathit{fibr}(y) \times \mathit{prop}(\mathit{fibr}(y)) \quad \text{avec} \quad \mathit{fibr}(y) = \Sigma x : A.\ f\ x = y$$

On note $A \approx B$ s'il existe une équivalence de A vers B.

L'axiome d'univalence de Voevodsky :
la fonction canonique $A = B \to A \approx B$ est une équivalence.
En conséquence, si $A \approx B$, alors $A = B$.

Formalise l'idée intuitive de raisonnement à isomorphisme près (pas toujours juste en théorie des ensembles).

Implique les axiomes d'extensionalité usuels : **FE**, **DFE**, **PE**.

Contenu calculatoire pas encore bien clair. ($\Rightarrow$ séminaire Th. Coquand)

HoTT pour les langages de programmation

La notion d'équivalence comme caractérisation très précise de ce qu'est un ≪bon≫ changement de représentation : pas juste une bijection entre deux types, mais une bijection qui ≪transporte≫ correctement les égalités.

Les types inductifs d'ordre supérieur (HIT) comme nouveau moyen pour programmer de manière ≪correcte par construction≫, moyen différent et complémentaire de la programmation à types dépendants.

Tout ce potentiel reste à réaliser : pas encore d'implémentation complète de HoTT + HIT ; des prototypes partiels dans Agda, Coq et Lean.

VI

Bibliographie

Bibliographie

Le grand livre de référence :

- *Homotopy Type Theory : Univalent Foundations of Mathematics*,
  The Univalent Foundations Program, Institute for Advanced Study, 2013,
  `https://homotopytypetheory.org/book/`
  En priorité : chapitres 1, 2, 3 + chapitre 6 pour les HITs.

Une présentation assez courte des HITs avec de bons exemples :

- Henning Basold, Herman Geuvers, Niels van der Weide : *Higher Inductive Types in Programming*. J. UCS 23(1) : 63-88 (2017).

Des bibliothèques pour travailler avec HoTT :

- En Agda : `https://github.com/HoTT/HoTT-Agda`
- En Coq : `https://github.com/HoTT/HoTT`
- En Lean : `https://github.com/leanprover/lean2/`