Théorie Homotopique des Types et Types Inductifs d'Ordre Supérieur

Florent Bréhard

juillet-août 2013

Table des matières

1 Introdution

Le paradoxe du barbier s'énonce ainsi : voici un barbier qui rase tous les hommes qui ne se rasent pas eux-mêmes et seulement ceux-ci. Se pose alors la question de savoir si ce barbier se rase lui-même ou non. Dans le premier cas, il n'appartient pas à la catégorie de ceux qui ne se rasent pas eux-mêmes, donc il ne devrait pas se raser. Mais dans le second cas, il ferait alors partie de cette catégorie et devrait donc se raser. Ainsi, d'une apparente définition anodine, on obtient une disjonction de deux cas desquels on peut dériver une contradiction. En théorie des ensembles naïve, on a une formulation similaire avec le paradoxe de Russell : soit $x = \{y \mid y \notin y\}$. Alors si $x \in x$, par définition $x \notin x$, mais si $x \notin x$, on a alors $x \in x$. Là aussi, on dérive une contradiction dans les deux cas.

A moins d'éluder le problème par une pirouette (le barbier pourrait être une femme), le logicien sérieux doit se poser les bonnes questions : de quels objets parle-t-on ? Quels axiomes ai-je le droit d'utiliser ? Quelles sont les règles de déduction logique à ma disposition ? Ce travail de fond, en amont des mathématiques classiques et à la frontière avec la philosophie, est celui de la formalisation de la logique et des processus de démonstration. Les mathématiques ne sont plus un simple raisonnement intuitif, mais deviennent un langage à part entière avec sa syntaxe et ses règles.

Alors que depuis l'Antiquité ce domaine faisait partie intégrante de la philosophie, c'est véritablement à la fin du XVIIIème siècle / début du XIXème siècle que cette discipline a connu son essor en tant que fondement rigoureux des mathématiques. Des mathématiciens comme Georg Cantor, Guiseppe Peano, Bertrand Russell ou David Hilbert ont ressenti le besoin de formaliser les mathématiques usuelles dont ils se servaient. Avec un langage rigoureux, comme celui de la logique du premier ordre, on a pu, à l'aide d'une dizaine d'axiomes, décrire une théorie suffisamment riche pour couvrir l'immense majorité des mathématiques : c'est la théorie des ensembles de Zermelo et Fraenkel, dite *ZF*. La notion de base est l'ensemble, avec le prédicat d'appartenance $\in$. Cette théorie prévaut encore aujourd'hui comme fondements (souvent implicites) des mathématiques. Elle répond notamment au paradoxe de Russell, en restreignant l'axiome de compréhension : il devient impossible de créer un ensemble $\{x \mid P(x)\}$ à partir d'un prédicat P, on admet en revanche $\{x \in y \mid P(x)\}$ où y est déjà un ensemble existant.

Néanmoins, cette approche de la formalisation des preuves a été jugée insuffisante par certains : qu'en est-il de la constructivité et de la calculabilité de ces preuves ? Par exemple, une preuve de l'existence d'un x tel que P fournit-elle effectivement un x en question ? Une preuve est-elle vérifiable par un *algorithme de décision* ? Peut-on automatiser cette vérification de preuves ? C'est avec l'apparition de la notion de *calculabilité*, puis carrément de l'informatique que l'on a pu apporter des réponses à ces questions. Le λ-calcul inventé par Alonzo Church puis étudié par de nombreux autres logiciens, comme Haskell Curry, est l'outil de base dans de nombreux cas : il permet de formaliser la notion de *programme*. La correspondance de Curry-Howard crée alors le véritable lien entre λ-calcul et production de preuve. Nous y reviendrons plus en détail.

Ce mémoire s'attache à l'étude d'une avancée plus récente dans le domaine. La théorie intuitioniste des types dépendants est en majeure partie due au mathématicien et philosophe suédois Per Martin-Löf, à partir des années 70. La plupart des *interactive theorems provers* comme *Coq* [2], Agda ou Isabelle, ont un système de types qui en est fortement inspiré. Plus récemment, la réinterprétation de ces concepts grâce la théorie de l'homotopie, issue de la topologie algébrique, promet de grandes avancées à venir. Vladimir Voevodsky y a beaucoup contribué, notamment en proposant l'*axiome d'univalence* qui identifie des objets entre lesquels il existe une équivalence. Très intéressante du point de vue mathématique, cette théorie apporte aussi un système de preuve entièrement constructif qui pourrait donner de nouvelles bases aux mathématiques, plus formalisables par informatique que les axiomes de ZF. Ce projet initié par Vladimir Voevodsky s'appelle *Univalent Foundations*, soutenu par l'*Institute for Advanced Study* (*IAS*) de Princeton, qui a organisé de nombreuses rencontres l'an passé sur le sujet et publié un livre de référence sur la théorie homotopique des types (HoTT) [4].

2 Generalites 1

2.1 le lambda-calcul

Le λ-calcul (ouvrage de référence : [1]) est un langage syntaxique auquel on ajoute des règles de calculs qui vont permettre de simuler le déroulement d'un calcul. On définit ainsi l'ensemble des λ-termes :

$$\Lambda ::= \mathcal{X} \mid (\Lambda\ \Lambda) \mid \lambda\mathcal{X} \cdot \Lambda \mid \mathcal{C}$$

où :

- $\mathcal{X}$ est un ensemble de variables : $x, y, z, t, \cdots$
- $(u\ v)$ est une application. Symboliquement, cela revient à considérer u appliqué à v. On omettra souvent les parenthèses, avec pour convention l'associativité à gauche.
- $\lambda x \cdot u$ est une λ-abstraction. Cela correspond à la fonction qui à x associe le terme u, u pouvant éventuellement contenir x. Pour un terme de la forme $\lambda x \cdot \lambda y \cdot u$, on écrira souvent $\lambda xy \cdot u$. Cela revient à dire que la fonction qui à x associe la fonction qui à y associe u peut être vue comme la fonction en 2 arguments qui à x et y renvoie u.
- $\mathcal{C}$ est un ensemble de constantes que l'on pourra définir au besoin. On peut les voir en quelque sorte comme des constructeurs avec des règles bien précises qui viendront enrichir la théorie.

On identifiera les termes modulo α-renommage, c'est à dire modulo renommage des variables liées par un λ. Ainsi $\lambda x \cdot u = \lambda y \cdot u[x \leftarrow y]$ pourvu que la variable y ne soit pas libre dans u.
On introduit maintenant la règle de calcul β :

$$\frac{}{(\lambda x \cdot u)v \to_\beta u[x \leftarrow v]}\ Subst$$

Ainsi que les règles de passage au contexte :

$$\frac{u \to_\beta u'}{u\ v \to_\beta u'v}\ App_1 \qquad \frac{v \to_\beta v'}{u\ v \to_\beta uv'}\ App_2 \qquad \frac{u \to_\beta u'}{\lambda x \cdot u \to_\beta \lambda x \cdot u'}\ Abstr$$

On notera $\to_\beta^*$ la clôture réflexive transitive de $\to_\beta$, et on dira que deux termes u et v sont convertibles si et seulement si il exite w avec $u \to_\beta^* w$ et $v \to_\beta^* w$. Par la suite, on identifiera les termes modulo convertibilité. A priori, on ne s'intéresera pas beaucoup au processus de normalisation des termes, qui peut se voir comme la normalisation d'une preuve (élimination des coupures).

2.2 Typage simple et correspondance de Curry-Howard

La théorie des types est une façon de concevoir les objets mathématiques comme étant chacun rangé dans une catégorie bien définie. C'est un point de vue très différent de la théorie des ensembles où la notion d'appartenance reste très générale. Par exemple, la construction des ordinaux pose problème en terme de types : comment un ensemble pourrait-il avoir comme élément une partie de lui-même ? Néanmoins, il est clair que dans les mathématiques usuelles, on raisonne intuitivement en terme de types. On ne se demande par exemple pas si $x \in 3$ ou si $\mathcal{M}(n, \mathbb{R}) \in \mathbb{N}$. Les types répondent justement à ce principe et sont bien plus efficaces à implémenter que les axiomes de ZF.

Voici dans un premier temps le typage simple en λ-calcul :
On dispose de types de base $\mathcal{B} = nat, bool, \cdots$
et on construit les types de manière inductive :

$$\mathcal{T} ::= \mathcal{B} \mid \mathcal{T} \to \mathcal{T}$$

Le typage des λ-termes s'effectue alors dans un environnement Γ constitué de *bindings* de la forme $(x : F)$ où $x \in \mathcal{X}$ et $F \in \mathcal{T}$, les x apparaissant devant être distincts deux à deux. L'environnement permet simplement d'attribuer un type aux variables libres des termes. On a lors les règles naturelles suivantes, qui font de ce λ-calcul la base des langages fonctionnels comme OCaml :

$$\frac{(x : F) \in \Gamma}{\Gamma \vdash x : F}\ Var \qquad \frac{\Gamma, x : F \vdash u : G}{\Gamma \vdash \lambda(x : F) \cdot u : F \to G} \to -Intro \qquad \frac{\Gamma \vdash u : F \to G \quad \Gamma \vdash v : F}{\Gamma \vdash u\ v : G} \to -Elim$$

Parmi les propriétés intéressantes du typage, on a que le type est préservé par β-réduction, et que tout terme typable dans ce système est fortement normalisant. La propriété de forte normalisation est à la fois importante et dérangeante. D'une part, elle garantit que le calcul termine, donc que tout est bien défini. La plupart des assistants de preuve tels *Coq* imposent la forte normalisation. Mais il en résulte par des théorèmes de calculabilité que notre système n'est plus Turing-complet. Néanmoins, en pratique cela importe peu, *Coq* permettant d'implémenter une très vaste classe de fonctions récursives totales.

Or de nombreux logiciens ont remarqué une analogie très forte entre les systèmes de types et les systèmes logiques. On appelle cela la correspondance de Curry-Howard [9]. Voici les ponts à faire entre ces deux mondes :

Types	$\longleftrightarrow$	Propositions
$F \to G$	$\longleftrightarrow$	$F \Rightarrow G$
$u : F$	$\longleftrightarrow$	u terme de preuve de F
Dérivation de typage	$\longleftrightarrow$	Dérivation d'une preuve
Normalisation d'un terme	$\longleftrightarrow$	Normalisation de la preuve, élimination de coupures

Avec ce système de types simples, on voit que l'on obtient l'équivalent d'une logique propositionnelle minimale. Elle sera dite intuitionniste, puisque toutes les tautologies (c'est-à-dire les formules booléennes pour lesquelles on a true dans toute la table de vérité) ne sont pas prouvables : seules les règles décrites plus haut sont autorisées.

Pour rendre cette logique plus expressive, ajoutons le type du "faux" : $\bot$. La négation de F se note alors $\neg F \equiv F \to \bot$. Touver un terme qui habite le type $\bot$, c'est trouver une contradiction. Il n'y a donc pas de règle d'introduction de $\bot$. Voici la règle d'élimination, qui traduit que l'on peut tout montrer à partir de $\bot$:

$$\frac{\Gamma \vdash u : \bot}{\Gamma \vdash u : F}\ \bot - Elim$$

2.3 Les types dépendants

Le système de types que nous venons de décrire n'est pas suffisamment riche pour exprimer la logique nécessaire aux mathématiques courantes. On peut apporter plusieurs extensions :

- On peut sans problème ajouter des constructions pour la conjonction $\wedge$, la disjonction $\vee$, etc. Il faut alors ajouter les règles d'introduction et d'élimination associées. On peut ainsi aller jusqu'à définir tous les opérateurs de la logique du premier ordre.
- Néanmoins, cela est insuffisant pour un assistant de preuve moderne. Par exemple, la cohérence de l'arithmétique de Peano d'ordre 1 ou le théorème de Goodstein sont des énoncés indécidables avec l'arithmétique de Peano d'ordre 1. D'autres systèmes de types, comme le système F, permettent de passer à l'ordre 2 en définissant des prédicats sur les prédicats d'ordre 1. On peut ainsi avoir un ordre n pour tout n fixé à l'avance. Mais là encore ce n'est pas satisfaisant. La cohérence de l'arithmétique de Peano d'odre n demeure indécidable par exemple.

Pour répondre à ce problème, le philosophe et logicien suédois Per Martin-Löf a proposé un nouveau système de types beaucoup plus puissant qui s'abstrait des limites d'ordre [7]. Il s'agit des types dépendants. On parle parfois de théorie intuitioniste des types dépendants. Ce système de types est à la base de la plupart des assistants de preuve d'aujourd'hui. Non seulement il est très expressif (par exemple, Thierry Coquand a expliqué que le système de types de Coq, basé sur les types dépendants, est le plus expressif possible avant l'incohérence), mais il a en plus l'avantage d'être entièrement constructif, contrairement aux usages de la théorie des ensembles. Ainsi, une preuve d'un énoncé existentiel fournira un témoin qui vérifie la propriété attendue. Autrement dit, toute preuve fournit un algorithme pour obtenir la valeur recherchée.

Nous allons d'abord présenter les concepts centraux de ce système de types, puis dans un second temps nous verrons comment elle a pu se réinterpréter en terme d'homotopie.

2.3.1 Les types comme valeurs de première classe

Pour pouvoir énoncer des prédicats dépendant eux-mêmes d'autres prédicats, il faut pouvoir manipuler les types comme des objets de première classe. Si F est un type comme on l'a défini plus haut, il doit lui même avoir un type. On note F : **Type**, où **Type** représenterait l'univers en quelque sorte.

Mais quel serait le type de **Type** alors ? En fait, on introduit toute une hiérarchie de types **Type** 0, **Type** 1, $\cdots$, **Type** n, $\cdots$., avec comme règle **Type** i : **Type** j pour $i < j$ et si A : **Type** i, alors A : **Type** j pour $j > i$. En pratique, on omettra cet indice et on notera simplement **Type** en gardant en mémoire cette hiérarchie.

Il devient alors possible d'écrire des types indexés par des éléments d'un autre type. Si A : **Type**, alors on peut construire le type $A \to$ **Type**. Un élément $P : A \to$ **Type** et une "fonction" qui à tout élément $a : A$ renvoie un type $P\ a$: **Type**. Cela s'écrit par la règle de formation de type suivante :

$$\frac{\Gamma \vdash A : \textbf{Type}\ i}{\Gamma \vdash A \to \textbf{Type}\ i : \textbf{Type}\ i} \to -Form$$

.
Les règles d'introduction et d'élimination restant les mêmes qu'auparavant.

On va maintenant ajouter de nouvelles façons de contruire des types afin d'enrichir notre système. On commence par le *produit dépendant*, qui servira de brique de base. On l'introduira sous forme de règles d'inférence qui s'ajouteront aux précédentes. Pour les types suivants, deux approches sont possibles :
– On peut ajouter des constructeurs dans $\mathcal{C}$, typés à l'aide des produits dépendants :
  1. un *constructeur de type* qui permet de former syntaxiquement le type désiré.
  2. un *constructeur d'éléments* pour créer des éléments dans ce type. C'est le même principe que les types construits d'*OCaml* (sans récursivité pour le moment), sauf qu'ici il faut considérer le type comme "*l'espace topologique engendré*" par ces constructeurs, là où en programmation classique on considère que ce type ne contient que ces éléments de base.
  3. un *principe d'induction* muni de *règles de calcul*, qui permet de définir des fonctions sur cet ensemble à partir des éléments de base définis par les constructeurs. Cela correspond aux fonctions en *OCaml* définis sur un type grâce au *pattern-matching*.
– On peut aussi introduire ces types directement via des règles d'inférence.
  1. Une *règle de formation du type* au sein de l'univers.
  2. Une *règle d'introduction* qui permet de créer des termes habitant ce type.
  3. Une *règle d'élimination* qui permet d'éliminer ce type, en créant des fonctions de ce type vers un autre type à partir des cas de base.
  4. Des *règles de calcul* liées à l'élimination (traitement des cas de base).

2.3.2 Le type produit dépendant

Le *produit dépendant* correspond à une notion étendue de fonction. Au lieu d'avoir une fonction de type $A \to B$, qui à $a : A$ associerait un élément dans un type B fixé, on voudrait que la valeur d'arrivée se situe dans un type $P\ a$ dépendant de a.

On a donc une règle de formation du produit dépendant :

$$\frac{\Gamma \vdash A : \mathbf{Type}\ i \quad \Gamma \vdash P : A \to \mathbf{Type}\ i}{\Gamma \vdash \mathbf{\Pi}(a : A), P\ a : \mathbf{Type}\ i}\ \mathbf{\Pi} - Form$$

et les règles d'introduction et d'élimination :

$$\frac{\Gamma, x : A \vdash h : P\ x}{\Gamma \vdash \lambda(a : A) \cdot h : \mathbf{\Pi}(x : A), P\ x}\ \mathbf{\Pi} - Intro \qquad \frac{\Gamma \vdash f : \mathbf{\Pi}(x : A), P\ x \quad \Gamma \vdash a : A}{\Gamma \vdash f\ a : P\ a}\ \mathbf{\Pi} - Elim$$

Le produit dépendant s'interprète très bien en logique des prédicats : puisqu'un élément qui habite un type T équivaut à une preuve de cette proposition T, un élément $f : \mathbf{\Pi}(x : A), P\ x$ est une fonction qui à tout élément $(a : A)$ renvoie un élément (donc une preuve) de $P\ a$. Ainsi, f est une preuve de $\forall x \in A, P(a)$.

Le produit dépendant est vraiment ce qui rend le système de types de Martin-Löf si puissant et expressif. On peut par exemple exprimer l'implication simple entre A et B donnée par le type $A \to B$ par : $\mathbf{\Pi}(a : A), B$ où le type dépendant $P : A \to \mathbf{Type}$ serait donné par $P :\equiv \lambda(a : A) \cdot B$.
Supposons maintenant que l'on ait un type A, une famille de types $B : A \to \mathbf{Type}$ et enfin une famille doublement indexée : $P : \mathbf{\Pi}(a : A), B\ a \to \mathbf{Type}$, alors on peut considérer le type :

$$T :\equiv \mathbf{\Pi}(a : A), \mathbf{\Pi}(b : B\ a), P\ a\ b$$

On le notera de manière plus compacte :

$$T :\equiv \mathbf{\Pi}(a : A)(b : B\ a), P\ a\ b$$

Si $f : \mathbf{\Pi}(a : A)(b : B\ a), P\ a\ b$, l'écriture $f\ a\ b$ est redondante, puisque b, de part son type $B\ a$, fournit le premier argument a. Dans le cas dépendant, où il n'y a pas d'ambiguité, on notera :

$$T :\equiv \mathbf{\Pi}\{a : A\}(b : B\ a), P\ a\ b$$

signifiant par là qu'on s'autorisera l'écriture $f\ b$ au lieu de $f\ a\ b$.

2.3.3 Le type somme dépendante

Soit $A : \mathbf{Type}$ et $P : A \to \mathbf{Type}$. On voudrait introduire une généralisation de la notion de produit cartésien dans lequel les paires $\langle x, u \rangle$ seraient constituées d'un élément $x : A$ et $u : P\ x$. On notera le type ainsi obtenu *somme dépendante*, notée :

$$\mathbf{\Sigma}(x : A), P\ x$$

qui se voit comme l'espace total obtenu par union disjointe des $P\ x$ pour $x : A$. En logique, ce type s'interprète comme la proposition : il existe $x : A$ tel que $P\ x$, et un élément $\langle x, u \rangle$ de ce type est constitué d'un $x : A$ avec une preuve u de $P\ x$. Ce procédé est donc entièrement constructif, car fournir une preuve de "il existe $x : A$ tel que $P\ x$" revient à exhiber un tel x et la preuve associée.

On peut donc introduire le constructeur de type :

$$\mathbf{\Sigma} : \mathbf{\Pi}(A : \mathbf{Type}), (A \to \mathbf{Type}) \to \mathbf{Type}$$

Ce qui revient à considérer la règle de formation du type somme dépendante :

$$\frac{\Gamma \vdash A : \mathbf{Type}\ i \quad \Gamma \vdash P : A \to \mathbf{Type}\ i}{\Gamma \vdash \mathbf{\Sigma}(a : A), P\ a : \mathbf{Type}\ i}\ \mathbf{\Sigma} - Form$$

On dispose naturellement du constructeur des paires dépendantes :

$$\mathsf{paird} : \mathbf{\Pi}\{A : \mathbf{Type}\}\{P : A \to \mathbf{Type}\}(a : A), P\ a \to \mathbf{\Sigma}(x : A), P\ x$$

mais on notera :

$$\langle a, u \rangle :\equiv \mathsf{paird}\ a\ u$$

On a de manière équivalente la règle :

$$\frac{\Gamma \vdash a : A \quad \Gamma \vdash u : P\ a}{\Gamma \vdash \langle a, u \rangle : \mathbf{\Sigma}(x : A), P\ x}\ \mathbf{\Sigma} - Intro$$

On considère maintenant le principe d'induction des paires dépendantes, qui dit que pour créer une fonction dépendante de $\mathbf{\Sigma}(x : A), P\ x$ vers une famille de types $D : (\mathbf{\Sigma}(x : A), P\ x) \to \mathbf{Type}$, il suffit de donner les valeurs sur les $w \equiv \langle a, u \rangle$, via une fonction :

$$d \equiv \lambda(x : A) \cdot \lambda(u : P\ x) \cdot g \quad : \mathbf{\Pi}(x : A)(u : P\ x), D\ \langle x, u \rangle$$

Le principe d'induction est alors :

$$\mathbf{ind}_{\Sigma(x:A),Px} : \mathbf{\Pi}(D : (\mathbf{\Sigma}(x : A), P\ x) \to \mathbf{Type})(d : \mathbf{\Pi}(x : A)(u : P\ x), D\ \langle x, u \rangle)(w : \mathbf{\Sigma}(x : A), P\ x), D\ w$$

ce qui fait qu'on a une fonction :

$$f :\equiv \mathbf{ind}_{\Sigma(x:A),Px}\ D\ d : \mathbf{\Pi}(w : \mathbf{\Sigma}(x : A), P\ x), D\ w$$

avec comme règle de calcul :

$$f\ \langle a, u \rangle :\equiv d\ a\ u$$

ce qui équivaut aux règles d'élimination et de réduction suivantes :

$$\frac{\Gamma \vdash D : (\mathbf{\Sigma}(x : A), P\ x) \to \mathbf{Type} \quad \Gamma \vdash d : \mathbf{\Pi}(x : A)(u : P\ x), D\ \langle x, u \rangle}{\Gamma \vdash \mathbf{ind}_{\Sigma(x:A),Px}\ D\ d : \mathbf{\Pi}(w : \mathbf{\Sigma}(x : A), P\ x), D\ w}\ \mathbf{\Sigma} - Elim$$

$$\frac{\begin{array}{c}\Gamma \vdash D : (\mathbf{\Sigma}(x : A), P\ x) \to \mathbf{Type} \quad \Gamma \vdash d : \mathbf{\Pi}(x : A)(u : P\ x), D\ \langle x, u \rangle \\ \Gamma \vdash a : A \quad \Gamma \vdash u : P\ a\end{array}}{\Gamma \vdash \mathbf{ind}_{\Sigma(x:A),Px}\ D\ d\ \langle a, u \rangle :\equiv d\ a\ u}\ \mathbf{\Sigma} - Red$$

En particulier, ce principe d'induction nous permet de définir les 2 fonctions de projection canoniques :

$$\mathsf{projd}_1 : (\mathbf{\Sigma}(x : A), P\ x) \to A$$

$$\mathsf{projd}_2 : \mathbf{\Pi}(w : \mathbf{\Sigma}(x : A), P\ x), P\ (\mathsf{projd}_1\ w)$$

telles que :

$$\mathsf{projd}_1\ \langle a, u\rangle \equiv a \quad \text{et} \quad \mathsf{projd}_2\ \langle a, u\rangle \equiv u$$

La permière se montrer simplement avec $D :\equiv \lambda w \cdot A$ et $d :\equiv \lambda au \cdot a$. La deuxième s'obtient ensuite avec $D :\equiv \lambda w \cdot P\ (\mathsf{projd}_1\ w)$ et $d :\equiv \lambda au \cdot u$.

2.3.4 Le type produit cartésien

Au vu de la construction du type somme dépendante que nous avons défini, le produit cartésien de (A : **Type**) et de (B : **Type**) devient un cas particulier de cette somme. Il suffit de prendre pour ($P : A \to$ **Type**) la famille indexée sur A constante à B :

$$P :\equiv \lambda x \cdot B$$

Dans ce cas précis, on notera le produit $A \times B :\equiv \mathbf{\Sigma}(a : A), B$ et les paires (a, b) au lieu de $\langle a, b\rangle$. Les projections sont alors :

$$\mathsf{proj}_1 : A \times B \to A \quad \text{et} \quad \mathsf{proj}_2 : A \times B \to B$$

Notons que lorsqu'on aura défini un "type à deux éléments", noté **2** ou **Bool**, en définissant $P :$ **2** $\to$ **Type** par $P\ 0_2 :\equiv A$ et $P\ 1_2 :\equiv B$, on pourra aussi voir $A \times B$ comme $\mathbf{\Pi}(b :$ **2**$), P\ b$.

2.3.5 Les types construits

Vu le caractère assez général de la création de nouveaux types en se basant sur les "briques élémentaires" que sont les produits dépendants, on est tenté d'inventer une méthode générale pour créer de nouveaux types. Les types inductifs répondent à celà, mais il existe diverses façons de les introduire. On les étudiera en détail plus tard, en attendant on se contente des types construits, qui en sont une version plus simple. En *Coq*, un nouveau type s'introduit par :

Inductive Foo : **Type** :=
| $\mathsf{constr}_1 : (type_1)$
| $\mathsf{constr}_2 : (type_2)$
| $\cdots$

où les constr_i sont des constructeurs qui vont engendrer ce type et qu'on peut ainsi ajouter à notre liste $\mathcal{C}$ pour enrichir notre λ-calcul. Il faut néanmoins se poser la question des types $type_i$ à donner à ces contructeurs. Comme ils doivent engendrer ce nouveau type, on va demander qu'ils soient de la forme :

$$T_1 \to \cdots \to T_n \to Foo$$

où les types $T_1 \cdots T_n$ (avec éventuellement $n = 0$) seront les types des arguments. On autorise n'importe quel type pour ces arguments, mais pour le moment on ne souhaite pas faire intervenir le type Foo lui-même (les types construits sont donc un cas particulier des types inductifs que l'on verra plus tard). La seule condition est d'avoir un type Foo en retour, ce qui traduit la crétion d'un nouvel élément de type Foo. Par exemple, pour $n = 0$, on définit un constructeur constant dans Foo, pour $n = 1$, on peut imaginer un constructeur de type : **Nat** $\to Foo$ (on obtiendrait des éléments de type Foo indexés par des entiers).

Il faut maintenant spécifier quelles sont les règles d'élimination du type, afin de pouvoir définir des fonctions sur ce type. En fait, c'est très simple : cela fonctionne exactement comme le *pattern-matching* d'*OCaml*. On fait en quelque sorte une disjonction de cas, ce qui revient à dire que le type ne contient que des éléments dont la forme est spécifiée par les constructeurs (à homotopie près bien sûr, ce ne sont pas des égalités définitionelles). Les principes de récurrence / d'induction vont donc prendre en argument, en plus du type d'arrivée B (ou de la fibration P dans le cas dépendant), une fonction pour chaque constructeur qui dit

comment construire l'image d'un élément obtenu par ce constructeur.

On va donner quelques exemples de types construits, pour enfin arriver au type "égalité" dit **paths** qui est à la base de HoTT. La notion d'inductivité apparaitra alors clairement dans chaque cas comme une "évidence".

le type Empty C'est le cas le plus extrême : on construit un type vide !

Inductive **Empty** : **Type** := .

Ainsi, trouver un élément de type **Empty** c'est obtenir une contradiction. Pour cette raison, on retrouve le type $\bot$ défini plus haut, et on le désignera par ce symbole, ou encore par **0**.
Le principe d'induction dit alors la chose suivante :
"pour définir une fonction de $\bot \to P$, il suffit de fournir... rien !"

$$\mathbf{ind}_\bot : \mathbf{\Pi}(P : \bot \to \mathbf{Type})(z : \bot), P\ z$$

d'où dans le cas non dépendant :

$$\mathbf{rec}_\bot : \mathbf{\Pi}(B : \mathbf{Type}), \bot \to B$$

En effet, le type $\bot$ est vide ! On en déduit alors la célèbre règle d'élimination du faux : sachant $\bot$, je peux prouver n'importe quelle proposition P, car le principe d'induction me donne $\bot \to P$ comme fonction, donc si j'ai un élément (une preuve) de $\bot$, j'obtiens par cette fonction une preuve de P.

le type Unit Ici, on souhaite définir un type à un seul élément, type que l'on note **Unit** ou tout simplement **1**. Son unique élément se note $\star$. On a :

Inductive **Unit** : **Type** :=
| $\star$: **Unit**

et le principe d'induction dit qu'il faut juste donner la valeur de $\star$:

$$\mathbf{ind}_1 : \mathbf{\Pi}(P : \mathbf{1} \to \mathbf{Type}), P\ \star \to (\mathbf{\Pi}(z : \mathbf{1}), P\ z)$$

le type Bool
Inductive **Bool** : **Type** :=
| 0_2 : **Bool**
| 1_2 : **Bool**

C'est la même chose avec deux éléments : un type **Bool** ou **2** avec les constructeurs 0_2 et 1_2. Le principe d'induction requiert de donner les valeurs sur 0_2 et 1_2 :

$$\mathbf{ind}_2 : \mathbf{\Pi}(P : \mathbf{2} \to \mathbf{Type}), P\ 0_2 \to P\ 1_2 \to (\mathbf{\Pi}(b : \mathbf{2}), P\ b)$$

On pourra grâce à ce principe montrer plus tard (une fois le type égalité **paths** défini) que les éléments 0_2 et 1_2 sont bien distincts, à savoir que le type **paths** 0_2 1_2 n'est pas habité.

le type coproduit On n'a pas encore défini de construction correspondant au "ou" $\vee$ logique. Voici comment définir une somme disjointe de deux types :

Inductive Coprod $(A\ B : \mathbf{Type}) : \mathbf{Type} :=$
$\quad | \ \mathsf{inl} : A \to \mathsf{Coprod}\ A\ B$
$\quad | \ \mathsf{inr} : B \to \mathsf{Coprod}\ A\ B$

On notera $A + B :\equiv \mathsf{Coprod}\ A\ B$.
Son principe d'induction exprime que ses éléments sont la réunion disjointe de ceux de A et de B :

$$\mathbf{ind}_{A+B} : \mathbf{\Pi}(P : A + B \to \mathbf{Type}), (\mathbf{\Pi}(a : A), P\ (\mathsf{inl}\ a)) \to (\mathbf{\Pi}(b : B), P\ (\mathsf{inr}\ b)) \to \mathbf{\Pi}(x : A + B), P\ x$$

Il y a une différence essentielle avec le $\vee$ en logique classique. Dans cette dernière, la proposition $A \vee \neg A$ est par exemple tout le temps vraie. Ce n'est pas le cas en logique intuitoniste. Dans notre système de type, avec le $\vee$ défini par le coproduit, donner une preuve (donc un élément) de $A + B$, c'est avoir soit un $\mathsf{inl}\ a$ soit un $\mathsf{inr}\ b$, donc une preuve de A ou un preuve de B. Cela réduit peut-être le nombre de théorèmes prouvables, mais a l'avantage de demeurer totalement constructif !

On peut aussi définir le coproduit à partir d'une somme dépendante indexée sur $\mathbf{2}$. Si $P : \mathbf{2} \to \mathbf{Type}$ est donné par :

$$P\ 0_2 :\equiv A \quad \text{et} \quad P\ 1_2 :\equiv B$$

alors $A + B$ correspond exactement à $\mathbf{\Sigma}(b : \mathbf{2}), P\ b$.

En bref, les types construits permettent de décrire facilement des types obtenus sans induction à partir d'autres types. On verra plus tard une généralisation : les types inductifs, puis les types inductifs d'ordre supérieur sur lesquels on s'attardera.

3 Bases de la théorie homotopique des types

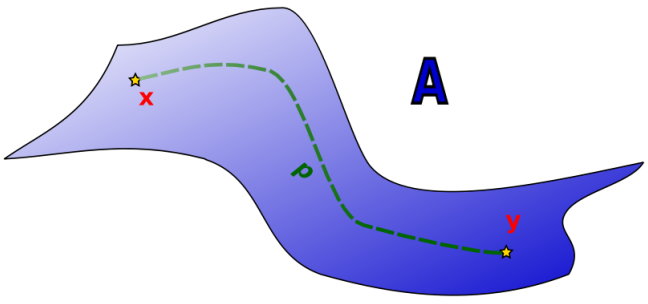
3.1 Le type paths

On définit le type inductif **paths**, qui s'interprète comme le type des *égalités* entre objets, ou encore comme les *chemins* entre ceux-ci. Il jouera un rôle crucial dans la théorie homotopique des types en lui donnant toute son expressivité :

Inductive **paths** $\{A : \mathbf{Type}\}(a : A) : A \to \mathbf{Type} :=$
$\quad | \ \mathsf{idpath} : \mathbf{paths}\ a\ a$

On simplifiera les notations ainsi :

$$x = y :\equiv \mathbf{paths}\ x\ y \quad \text{et} \quad 1_x :\equiv \mathsf{idpath}\ x$$



Le principe d'induction est le suivant (il est noté en général **J** au lieu de $\mathbf{ind}_{paths}$) :

$$\begin{aligned}\mathbf{J} : \quad & \mathbf{\Pi}\{A : \mathbf{Type}\}(D : \mathbf{\Pi}(x\ y : A), x = y \to \mathbf{Type})(d : \mathbf{\Pi}(x : A), D\ x\ x\ 1_x), \\ & \mathbf{\Pi}(x\ y : A)(p : x = y), D\ x\ y\ p\end{aligned}$$

avec comme règle de calcul :

$$\text{si } f :\equiv \mathbf{J}\ D\ d \quad \text{alors pour tout } x : A, \quad f\ x\ x\ 1_x \equiv d\ x$$

Ce principe dit que pour obtenir une fonction de type D (donc une preuve de D), il suffit de traiter le cas $x \equiv y$ et $p \equiv 1_x$. C'est ce principe qui permettra de montrer la plupart des théorèmes que nous énoncerons. Voici un premier exemple. Ayant un élément $p : x = y$ (donc un chemin de x vers y), on voudrait de manière naturelle pouvoir définir son inverse. Il nous faut donc une fonction qui à p lui associe son inverse. Cela revient à dire que l'égalité est une relation symétrique. Soit donc :

$$\text{inverse} : \mathbf{\Pi}\{A : \mathbf{Type}\}\{x\ y : A\}, x = y \to y = x$$

(avec comme notation $p^{-1} :\equiv \text{inverse}\ p$), que l'on obtient par le principe d'induction **J** avec :

$$D\ x\ y\ p :\equiv y = x \quad \text{et} \quad d\ x :\equiv 1_x.$$

On a en particulier, grâce à la règle de calcul, que pour tout $x : A$, $\text{inverse}\ 1_x = 1_x$.

On doit alors se poser une question essentielle : à quel moment l'interprétation de ce système de types en terme d'homotopie devient-il significatif ? En effet, on pourrait à première vue se dire que voir le type égalité **paths** comme celui des chemins est tout à fait artificiel, et qu'en réalité les seuls habitants de ce type sont ceux que l'on a spécifiés via le constructeur idpath. Ainsi, les seuls chemins sont les chemins triviaux (dits réflexifs), et un type peut donc se concevoir comme une collection de points, c'est-à-dire grosso modo comme un ensemble.

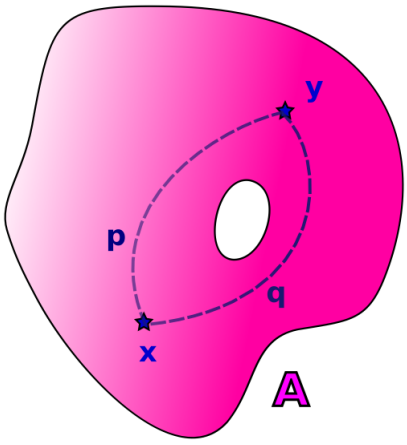
Mais dans ce cas, on serait alors en mesure de prouver la proposition :

$$\mathbf{\Pi}(A : \mathbf{Type})(x\ y : A)(p\ q : x = y), p = q$$

qui par induction de chemins sur q se réduit en :

$$\mathbf{\Pi}(A : \mathbf{Type})(x : A)(p : x = x), p = 1_x$$

Or ce principe, nommé *UID* (*Uniqueness of Identity Proofs*) (ou encore *axiome K* dans le cas des boucles), n'est pas démontrable en règle générale, pour tout type A. Avec l'axiome d'univalence que nous introduirons après, on pourra même exhiber des contre-exemples simples. En revanche, certains types ont effectivement cette propriété, comme ceux à "1 seul élément" (comme **Unit**, donc une preuve $\mathbf{\Pi}(x\ y : A), x = y$), ou ceux dont l'égalité est décidable ($\mathbf{\Pi}(x\ y : A), (x = y) \vee \neg(x = y)$).



Lorsque les théoriciens des types ont compris que l'interprétation des types comme des *ensembles* était insatisfaisante, ils ont adopté un autre point de vue : celui de les voir comme des *espaces topologiques*, et l'égalité entre deux points comme un chemin de l'un vers l'autre. Un chemin peut être imaginé comme une fonction continue de l'intervalle [0 :1] vers le type, avec comme extrémités x et y. Cependant, il faut bien garder à l'esprit que le chemin est une notion primitive, en aucun cas on ne s'intéressera à ce que pourraient être les "points intermédiaires" du chemin entre x et y.

Cette approche est nettement plus puissante pour aborder cette théorie des types, les liens qui unissent les deux sont très forts. Ainsi, on sera par exemple capable de montrer des résultats d'homotopie en *Coq* rien qu'avec ce système de types. À l'inverse, la théorie de l'homotopie et l'intuition géométrique se révèleront précieux lors de certaines preuves ou constructions.

3.2 La structure de groupoïde du type paths

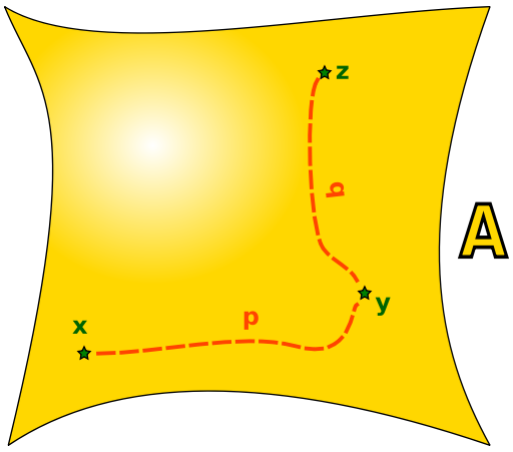
L'idée naturelle avec les chemins est de définir une fonction de concaténation :

$$\mathsf{concat} : \mathbf{\Pi}\{A : \mathbf{Type}\}\{x\ y\ z : A\}, x = y \to y = z \to x = z$$

avec la notation infixe :

$$p \centerdot q :\equiv \mathsf{concat}\ p\ q$$

Démonstration.
Par induction sur p puis sur q, il suffit de le montrer pour $x \equiv y \equiv z$ et $p \equiv q \equiv 1_x$, 1_x convient alors. Du coup, on a bien $1_x \centerdot 1_x \equiv 1_x$, mais en général on n'a pas $1_x \centerdot p \equiv p$ et $p \centerdot 1_y \equiv p$. □



Voici ensuite les théorèmes qui confèrent au type **path** $x\ y$ une structure de groupoïde (on fixe en paramètre un A : **Type**, des points $x\ y\ z\ u : A$, et des chemins $p : x = y$, $q : y = z$, $r : z = u$) :

concat_1p p :	$1_x \centerdot p = p$
concat_p1 p :	$p \centerdot 1_y = p$
concat_pp_p $p\ q\ r$:	$(p \centerdot q) \centerdot r = p \centerdot (q \centerdot r)$
concat_pV p :	$p \centerdot p^{-1} = 1_x$
concat_Vp p :	$p^{-1} \centerdot p = 1_y$
inverse_V p :	$(p^{-1})^{-1} = p$
inverse_pp $p\ q$:	$(p \centerdot q)^{-1} = q^{-1} \centerdot p^{-1}$

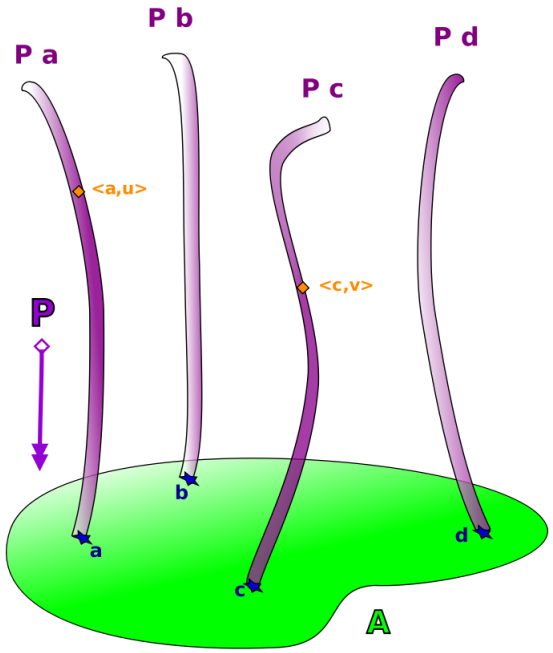
Démonstration.
par induction facile sur les chemins □

3.3 Fibrations, Sections et Fonctorialité

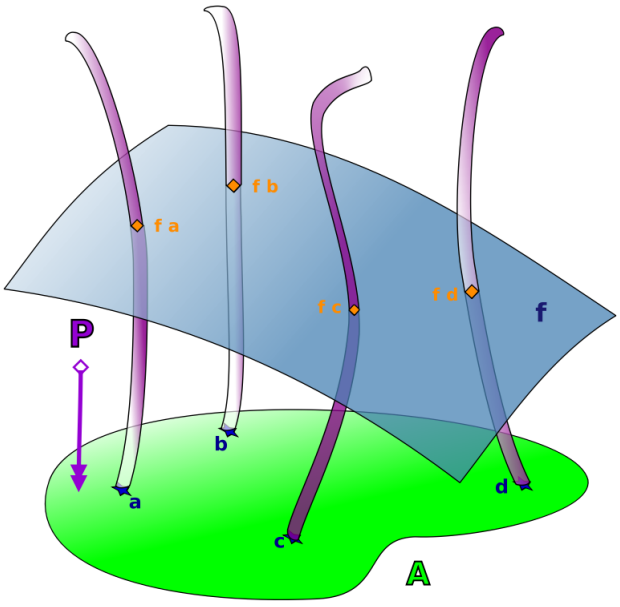
3.3.1 Interprétation homotopique des objets de base

On a vu qu'en λ-calcul classique, un élément $f : A \to B$ s'interprète comme une fonction de *domaine* A et de *codomaine* B, ce qui semble naturel. Qu'en est-il dans le cas dépendant ? Il faut d'abord avoir une représentation de ce qu'est un type dépendant.

Construire un type dépendant $P : A \to \mathbf{Type}$, c'est considérer un espace $P\ a$ au-dessus de chaque $a : A$. Cela correspond intuitivement à la notion de *fibration* en topologie algébrique : chaque espace $P\ a$ est une *fibre* au-dessus de a. On dit que $A : \mathbf{Type}$ est l'*espace de base*, $P : A \to \mathbf{Type}$ est la *fibration*, les $P\ a$ sont les *fibres* et $\mathbf{\Sigma}(a : A), P\ a$ est l'*espace total*, constitué des paires dépendantes $\langle a, u \rangle$ où $a : A$ est un point de l'espace de base, et $u : P\ a$ est un point dans la fibre $P\ a$ au-dessus de a. Dans le cas où $P \equiv \lambda(a : A) \cdot B$ avec $B : \mathbf{Type}$ fixé, sans variables libres, on parle de *fibration constante*.



Une fonction dépendante $f : \mathbf{\Pi}(a : A), P\ a$ s'appelle une *section*. C'est comme une fonction sauf que le codomaine n'est pas un type B constant. Au lieu de ça, à tout $a : A$, on associe une image $b : P\ a$.



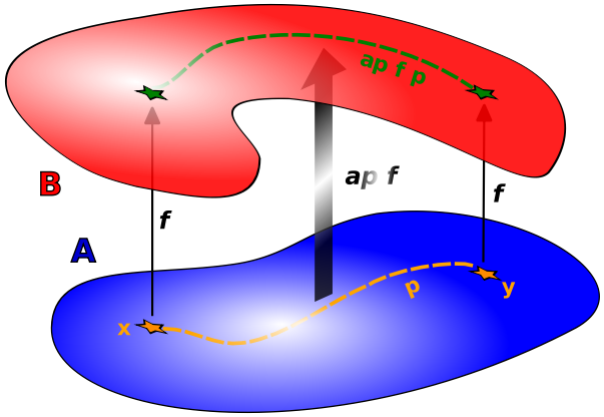
3.3.2 Fonctorialité dans le cas non-dépendant

Un autre point essentiel est qu'en HoTT, tous les objets : fibrations / fonctions / sections sont considérés comme *continus*. Certes, cela n'a a priori pas beaucoup de sens, puisqu'à aucun moment nous n'avons défini de topologie pour l'Univers. Mais il s'agit juste d'une vue de l'esprit. Bien sûr, si l'on cherche un modèle à HoTT, il faudra interpéter toutes les éléments du type fonction comme des fonctions continues. Néanmoins, cette notion de continuité n'est pas absurde. Les seuls "éléments topologiques" primitifs dont nous disposons sont les chemins. La caractérisation d'une fonction continue serait donc une fonction qui relève un chemin de x à y dans A en un chemin de $f\ x$ à $f\ y$ dans B. Et nous avons en effet cette propriété de fonctorialité des fontions sur les chemins (ap pour *action on paths*) :

$$\mathsf{ap} : \mathbf{\Pi}\{A\ B : \mathbf{Type}\}(f : A \to B)\{x\ y : A\}, x = y \to f\ x = f\ y$$

Démonstration.
par induction sur p : si $x \equiv y$ et $p \equiv 1_x$, il suffit de renvoyer 1_{fx}. □

Du point de vue logique, cela caractérise le fait que la relation d'égalité est une *relation de congruence* pour les fonctions. En terme d'homotopie, on a relevé $p : x = y$ en $\mathsf{ap}\ f\ p : f\ x = f\ y$.



La fonctorialité respecte les lois de groupe à la fois pour les chemins (concaténation) et les fonctions (composition) :

$$\begin{array}{ll}
\mathsf{ap_pp}\ f\ p\ q & : \mathsf{ap}\ f\ (p\ \centerdot\ q) = \mathsf{ap}\ f\ p\ \centerdot\ \mathsf{ap}\ f\ q \\
\mathsf{ap_V}\ f\ p & : \mathsf{ap}\ f\ p^{-1} = (\mathsf{ap}\ f\ p)^{-1} \\
\mathsf{ap_id}\ p & : \mathsf{ap}\ \mathsf{id}_A\ p = p \\
\mathsf{ap_compose}\ f\ g\ p & : \mathsf{ap}\ (g \circ f)\ p = \mathsf{ap}\ g\ (\mathsf{ap}\ f\ p)
\end{array}$$

3.3.3 Fonctorialité dans le cas dépendant

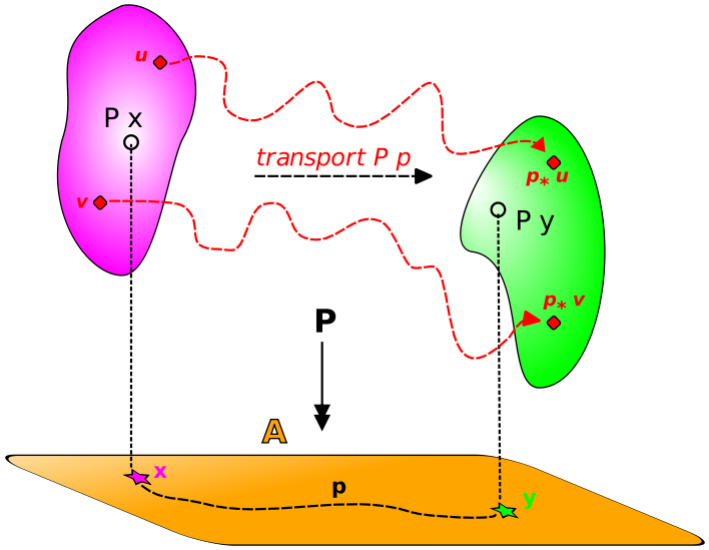
Dans le cas dépendent, c'est-à-dire avec une fibration non constante, on aimerait appliquer le même procédé. Soient $A : \mathbf{Type}$ l'espace de base, $P : A \to \mathbf{Type}$ la fibration et $f : \mathbf{\Pi}(a : A), P\ a$ une section. Supposons donné un chemin $p : x = y$, on aimerait en construire un de manière naturelle entre $f\ x$ et $f\ y$. Seulement, $f\ x : P\ x$ et $f\ y : P\ y$ ne vivent pas dans le même type, donc on ne peut directement parler de chemin entre eux.

Pourtant, par ce qu'on a vu avant, comme $x = y$ dans A, $P\ x = P\ y$ dans $\mathbf{Type}$, et donc les éléments de la première fibre devraient pouvoir "se transporter" dans la seconde. On obtient ainsi la notion très importante de *transport* :

$$\mathsf{transport} : \mathbf{\Pi}\{A : \mathbf{Type}\}(P : A \to \mathbf{Type})\{x\ y : A\}(p : x = y), P\ x \to P\ y$$

Démonstration.
par induction sur p, on se ramène au cas où $x \equiv y$ et $p \equiv 1_x$. On renvoie alors id_{Px}. □

Si le contexte est clair quant à la fibration considérée, on pourra noter $p_\star\ u :\equiv \mathsf{transport}\ P\ p\ u$.



En logique, le transport permet de montrer que notre type égalité correspond à l'égalité au sens de Leibniz, à savoir que deux éléments x et y sont égaux si et seulement si pour toute propriété P, $P\ x \Rightarrow P\ y$. Dans un sens, le transport permet d'habiter la propriété $P\ y$ si $x = y$ et si $P\ x$ est habitée. Dans l'autre sens, il suffit de prendre $P \equiv \lambda(a : A) \cdot x = a$.

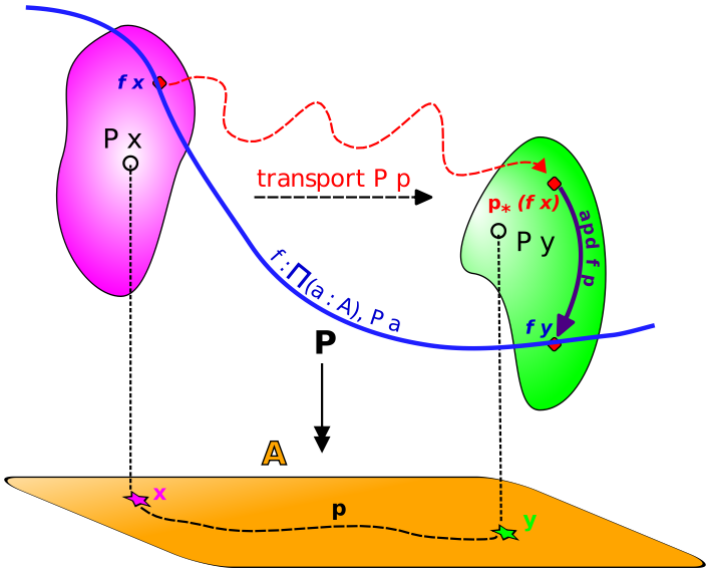
Si $u : P\ x$ et $v : P\ y$, on définit $u =^P_p v$ l'espace des chemins dépendants de u à v par :

$$u =^P_p v :\equiv \mathsf{transport}\ P\ p\ u = v$$

On peut alors relever $p : x = y$ en un chemin dépendant dans $f\ x =^P_p f\ y$:

$$\mathsf{apd} : \mathbf{\Pi}\{A : \mathbf{Type}\}\{P : A \to \mathbf{Type}\}(f : \mathbf{\Pi}(a : A), P\ a)\{x\ y : A\}(p : x = y), f\ x =^P_p f\ y$$

Démonstration.
par induction sur p, comme $(f\ x =^P_{1_x} f\ x) \equiv (f\ x = f\ x)$, il suffit de renvoyer 1_{fx}. □



Maintenant, comme ce sera souvent le cas, il faut vérifier que les définitions dans le cas non-dépendant peuvent se voir comme un cas particulier du cas dépendant, c'est-à-dire se ramener au cas d'une fibration constante. Ici, on veut relier $\mathsf{ap}\ f\ p$ et $\mathsf{apd}\ f\ p$. Voici d'abord un énoncé qui caractérise le transport dans

une fibration constante :

$$\begin{aligned}&\mathsf{transport}^{\mathsf{const}} : \quad \mathbf{\Pi}\{A\ :\ \mathbf{Type}\}(B : \mathbf{Type})\{x\ y : A\}(p : x = y)(b : B),\\ &\qquad\qquad \mathsf{transport}\ (\lambda(a : A) \cdot B)\ p\ b = b\end{aligned}$$

et donc :

$$\begin{aligned}&\mathsf{apd}^{\mathsf{const}} : \quad \mathbf{\Pi}\{A : \mathbf{Type}\}\{P : A \to \mathbf{Type}\}(f : \mathbf{\Pi}(a : A), P\ a)\{x\ y : A\}(p : x = y),\\ &\qquad\qquad \mathsf{apd}\ f\ p = \mathsf{transport}^{\mathsf{const}}\ B\ p\ (f\ x)\ \centerdot\ \mathsf{ap}\ f\ p\end{aligned}$$

3.3.4 Chemins dans l'espace total

On se place toujours dans un espace de base $A : \mathbf{Type}$ muni d'une fibration $P : A \to \mathbf{Type}$. On souhaite pouvoir caractériser les chemins de l'espace total $\mathbf{\Sigma}(a : A), P\ a$. S'il est vrai que si $w, w' : \mathbf{\Sigma}(a : A), P\ a$, alors $w = w' \Rightarrow \mathsf{projd}_1\ w = \mathsf{projd}_1\ w'$, il est en revanche faux de penser que $\langle a, u\rangle = \langle a, v\rangle \Rightarrow u = v$.

En fait, dans l'espace total $\mathbf{\Sigma}(a : A), P\ a$, l'espace des chemins $w = w'$ s'identifie à $\mathbf{\Sigma}(p : \mathsf{projd}_1\ w = \mathsf{projd}_1\ w'), p_\star\ (\mathsf{projd}_2\ w) = \mathsf{projd}_2\ w'$ grâce aux fonctions :

$$\mathsf{totalspace\_paths} : \mathbf{\Pi}\{w\ w' : \mathbf{\Sigma}(a : A), P\ a\}, w = w' \to \mathbf{\Sigma}(p : \mathsf{projd}_1\ w = \mathsf{projd}_2\ w'), p_\star\ (\mathsf{projd}_2\ w) = \mathsf{projd}_2\ w'$$

$$\mathsf{totalspace\_paths}^{-1} : \mathbf{\Pi}\{w\ w' : \mathbf{\Sigma}(a : A), P\ a\}, (\mathbf{\Sigma}(p : \mathsf{projd}_1\ w = \mathsf{projd}_2\ w'), p_\star\ (\mathsf{projd}_2\ w) = \mathsf{projd}_2\ w') \to w = w'$$

Démonstration.

- construction de $\mathsf{totalspace\_paths}$:
  Par induction sur le chemin dans $w = w'$, on est ramené à $w \equiv w'$ et alors on renvoie $\langle 1_{\mathsf{projd}_1 w}, 1_{\mathsf{projd}_2 w'}\rangle$, puisque $1_{\mathsf{projd}_1 w_\star}\ \mathsf{projd}_2\ w \equiv \mathsf{projd}_2\ w$.
- construction de $\mathsf{totalspace\_paths}^{-1}$:
  En effectuant une première induction sur w et w', on casse ces éléments en des paires $\langle w_1, w_2\rangle$ et $\langle w_1', w_2'\rangle$. On veut alors une fonction :

  $$(\mathbf{\Sigma}(p : w_1 = w_1'), p_\star\ w_2 = w_2') \to \langle w_1, w_2\rangle = \langle w_1', w_2'\rangle$$

  Par induction sur le $\mathbf{\Sigma}$-type, on en fait une fonction dépendante, puis par induction sur p et sur q enfin, on est ramené à :

  $$\langle w_1, w_2\rangle = \langle w_1, w_2\rangle$$

  où il suffit de renvoyer le chemin réflexif.
- preuve de $(\mathsf{totalspace\_paths}^{-1} \circ \mathsf{totalspace\_paths})\ p = p$:
  En cassant w et w' en $\langle w_1, w_2\rangle$ et $\langle w_1', w_2'\rangle$, puis par induction sur p, $p \equiv 1_{\langle w_1, w_2\rangle}$, et alors on a vu que $\mathsf{totalspace\_paths}$ envoie $1_{\langle w_1, w_2\rangle}$ sur $\langle 1_{\mathsf{projd}_1\langle w_1, w_2\rangle}, 1_{\mathsf{projd}_2\langle w_1, w_2\rangle}\rangle \equiv \langle 1_{w_1}, 1_{w_2}\rangle$, lui-même envoyé par $\mathsf{totalspace\_paths}^{-1}$ sur $1_{\langle w_1, w_2\rangle}$. On a donc la preuve par $1_{1_{\langle w_1, w_2\rangle}}$.
- preuve de $(\mathsf{totalspace\_paths} \circ \mathsf{totalspace\_paths}^{-1})\ \langle p, q\rangle = \langle p, q\rangle$:
  Par induction sur p et q, on a $\langle p, q\rangle \equiv \langle 1_{w_1}, 1_{w_2}\rangle$, et on applique les mêmes règles de calcul qu'avant, pour conclure avec $1_{\langle 1_{w_1}, 1_{w_2}\rangle}$.

□

On va voir que cela définit une *équivalence*, et l'on notera :

$$(w = w') \ \simeq\ \mathbf{\Sigma}(p : \mathsf{projd}_1\ w = \mathsf{projd}_1\ w'), p_\star\ (\mathsf{projd}_2\ w) = \mathsf{projd}_2\ w'$$

3.4 Structure de groupoïde d'ordre supérieur

La formidable richesse de HoTT vient du fait qu'on a une structure de groupoïde à chaque dimension de chemins. On peut très bien considérer que les théorèmes que nous avons énoncés font de l'espace des chemins une structure de groupoïde, sans plus en dire. Mais les égalités données sont toujours à homotopie près, donc ces théorèmes introduisent des chemins d'ordre 2, sur lesquels on peut énoncer des théorèmes qui vont à leur tour introduire des chemins d'ordre 3, etc... Voilà pourquoi on parle d'∞-groupoïde.

Pour autant, il est faux de penser que les espaces de chemins gardent la même structure en montant dans les dimensions. Plus la dimension est élevée, plus on peut exprimer de nouveaux théorèmes qui viennent enrichir la structure. Par exemple, en dimension 0 (donc l'espace de base A : **Type**), on n'a aucune structure de groupe. Avec les chemins d'ordre 1 ($x = y$ pour $x\ y : A$) apparait la concaténation qui forme la première opération de groupe. Les chemins d'ordre 2 peuvent aussi être concaténés, mais on a en plus une nouvelle opération, parfois appelée *concaténation horizontale*. Si A : **Type**, $x\ y\ z : A$, $p\ p' : x = y$, $q\ q' : y = z$, $r : p = p'$, $s : q = q'$, alors (par induction facile) :

$$\mathsf{concat}^2\ r\ s\ \ (\mathsf{notation} : \mathsf{r}\ \mathbf{..}\ \mathsf{s})\ \ :\ \ p\ \mathbf{.}\ q = p'\ \mathbf{.}\ q'$$

On a de plus une sorte de "distributivité" sur la concaténation classique :

$$\mathsf{concat\_concat}^2\ r\ r'\ s\ s'\ : (r\ \mathbf{..}\ s)\ \mathbf{.}\ (r'\ \mathbf{..}\ s') = (r\ \mathbf{.}\ r')\ \mathbf{..}\ (s\ \mathbf{.}\ s')$$

3.5 Équivalences et axiomes fondamentaux

3.5.1 Homotopies

En mathématiques, il est usuel de caractériser les fonctions par les valeurs qu'elles prennent en chaque point de leur domaine. En théorie des ensembles, cela est tout à fait naturel puisqu'une fonction est justement définie par un graphe. En théorie des types en revanche, la notion de fonction est primitive. Deux fonctions égales renvoient la même valeur en chaque point du domaine, mais la réciproque n'a aucune raison d'être vraie. On énoncera bientôt l'*axiome d'extensionnalité* qui permettra de faire cette identification. En attendant, on dira juste que ces deux fonctions sont homotopes, et on notera le type des homotopies, pour deux fonctions $f\ g : \mathbf{\Pi}(a : A), P\ a$:

$$(f == g) :\equiv \mathbf{\Pi}(x : A), f\ x = g\ x$$

Cette définition peut sembler étrange du point de vue de la théorie classique de l'homotopie, puisque dans ce cas, pour parler d'homotopie, il faudrait que le choix des chemins de $f\ x$ à $g\ x$ soit *continu* en $x : A$. Mais rappelons qu'en HoTT, toute fonction est vue comme continue. Ainsi, le produit dépendant que nous avons défini réalise un "choix continu" de chemins entre les images de f et g.

La relation d'homotopie est clairement une relation d'équivalence, et pour $f\ g\ h : A \to B$, nous noterons ainsi ces théorèmes :

$$\begin{array}{ll} \mathsf{funhom\_refl}\ f : & f == f \\ \mathsf{funhom\_sym}\ f\ g : & (f == g) \to (g == f) \\ \mathsf{funhom\_trans}\ f\ g\ h : & (f == g) \to (g == h) \to (f == h) \end{array}$$

Cette relation d'homotopie est également compatible avec la fonctorialité : si f et g de domaine A sont homotopes par une homotopie H, alors leur action sur un chemin $p : x = y$ dans A est liée par la relation suivante, appelée naturalité de ap :

$$\mathsf{ap\_naturality}\ H\ p : \mathsf{ap}\ f\ p\ \mathbf{.}\ H\ y = H\ x\ \mathbf{.}\ \mathsf{ap}\ g\ p$$

Et dans le cas dépendant :

$$\text{apd\_naturality } H\ p : \text{apd } f\ p \ \centerdot\ H\ y = \text{ap } (\text{transport } \_\ p)\ (H\ x) \ \centerdot\ \text{apd } g\ p$$

3.5.2 Équivalences

La notion d'équivalence permet d'identifier deux types via une fonction "bijective" entre les 2. Il existe différentes manières de définir les équivalences. Vladimir Voevodsky les a définies comme des fonctions de $A \to B$ dont les fibres sont *contractiles*. La fibre en $b : B$ de f correspond essentiellement aux antécédants de b par f, le tout vu à homotopie près, comme toujours. On écrit :

$$\text{hfiber } f\ b \ :\equiv\ \mathbf{\Sigma}(a : A), f\ a = b$$

Ce sont donc les paires $\langle a, p\rangle$ où $a : A$ et p est une preuve de $f\ a = b$. La notion de *contractibilité* est :

$$\text{iscontr } (A : \mathbf{Type}) : \mathbf{\Sigma}(a : A), \mathbf{\Pi}(x : A), x = a$$

qui donne donc un centre $a : A$ de rétraction et une preuve que chaque $x : A$ se rétracte vers a, et là encore ce choix de chemins doit être vu comme étant continu. La définition de Voevodsky est donc :

$$\text{isequiv } f \ :\equiv\ \mathbf{\Pi}(b : B), \text{iscontr } (\text{hfiber } f\ b)$$

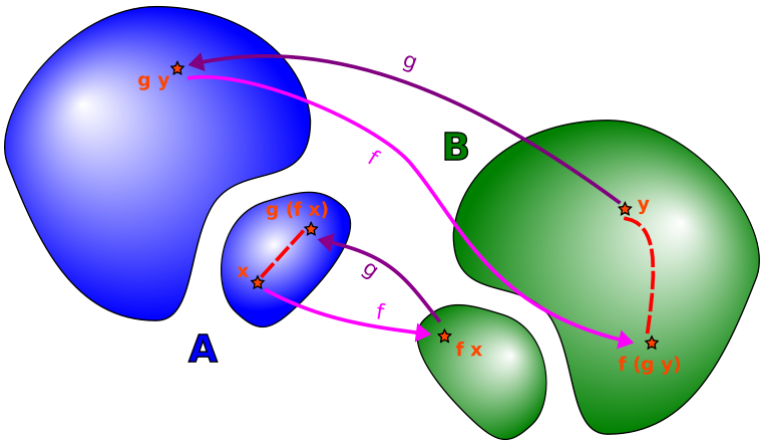
ce qui signifie que chaque fibre est habitée (car on a au moins le centre de rétraction) et se rétracte en un point, ce qui correspond, à homotopie près, à la notion de surjectivité et d'injectivité.

Une autre définition, équivalente mais souvent plus pratique, est de fournir pour $f : A \to B$ un "inverse" $g : B \to A$ tel que :

$$g \circ f == \mathrm{id}_A \quad \text{et} \quad f \circ g == \mathrm{id}_B$$

C'est cette définition de isequiv f qui sera le plus souvent utilisée :

$$\text{isequiv } f \ :\equiv\ \mathbf{\Sigma}(g : B \to A), ((g \circ f == \mathrm{id}_A) \times (f \circ g == \mathrm{id}_B))$$



Parfois, dans certains buts, on peut souhaiter que le type isequiv f se comporte mieux. En effet, il n'y a pas unicité pour le choix de l''inverse" g, et le type isequiv f n'est alors pas nécésairement contractile ("il n'est pas réduit à un seul élément comme **Unit**"). On admettra donc que l'on peut donner une autre définition de isequiv tel que si isequiv f est habité, alors il ne contient qu'un seul élément, à homotopie près bien sûr.

On peut maintenir définir l'équivalence de deux types comme étant l'existence d'une fonction d'équivalence entre eux :

$$(A \simeq B) \ :\equiv\ \mathbf{\Sigma}(f : A \to B), \text{isequiv } f$$

Si $f : A \simeq B$, on s'autorisera à utiliser f comme la fonction qui réalise cette équivalence, et on notera respectivement f^{inv}, f^{sect} et $f^{retract}$ l'inverse de f, et les preuves de section / rétraction.

Là encore, la notion d'équivalence est une relation d'équivalence au sens usuel :

réflexivité :	$\mathsf{id}_A : A \simeq A$
symétrie :	si $f : A \simeq B$ admet $g : B \to A$ comme inverse, alors $g : B \simeq A$
transitivité :	si $f : A \simeq B$ et $g : B \simeq C$, alors $g \circ f : A \simeq C$.

Les démonstrations sont triviales.

3.5.3 Axiomes fondamentaux

L'extensionnalité des fonctions Comme en mathématiques usuelles, on voudrait pouvoir dire que deux fonctions prenant les mêmes valeurs en chaque point de leur domaine sont égales. Remarquons que c'est un souhait plus fort que la simple η-conversion, car on peut imaginer deux fonctions qui renvoient les mêmes résultats, mais après des calculs différents. Par exemple, l'identité sur les entiers et la fonction qui renvoie le prédécesseur du successeur : bien que renvoyant les mêmes résultats, elles ne sont pas à proprement parler égales, fût-ce à $\beta\eta$-conversion près. Ainsi, une homotopie $H : f == g$ n'implique pas $f = g$, mais le contraire est bien entendu vrai :

$$\mathsf{happly}\ (f\ g : \mathbf{\Pi}(a : A), P\ a)\ :\ (f = g) \to (f == g)$$

Il suffit de raisonner par induction sur le chemin de $f = g$ pour pouvoir supposer $f \equiv g$ et prendre l'homotopie triviale $\lambda(a : A) \cdot 1_{fa}$.

Pour obtenir l'autre sens, on introduit l'axiome d'extensionnalité fonctionnelle qui fait de $\mathsf{happly}\ f\ g$ une équivalence, dont la réciproque est notée :

$$\mathsf{funext}\ :\ \mathbf{\Pi}\{f\ g : \mathbf{\Pi}(a : A), P\ a\}, (f == g) \to (f = g)$$

Nous voulons de plus les règles de calcul (propositonnelles) suivantes :

$$\mathbf{\Pi}(H : f == g), \mathsf{happly}\ (\mathsf{funext}\ H)\ x = H\ x$$

$$\mathbf{\Pi}(p : f = g), \mathsf{funext}\ \lambda(x : A) \cdot (\mathsf{happly}\ p\ x)\ =\ p$$

L'axiome d'extensionnalité fonctionnelle, qui n'est pas prouvable en tant que tel en HoTT, n'est néanmoins pas contradictoire et sera admis la plupart du temps. Ce sera en fait une conséquence de l'*axiome d'univalence*.

L'axiome d'univalence C'est l'axiome primordial en HoTT, il est dû à Vladimir Voevodsky. Il stipule que si deux types sont équivalents, alors ils sont égaux. Un sens est déjà clair : si deux types sont égaux, alors ils sont équivalents, par une fonction d'équivalence canonique, qui peut être vue comme une règle d'élimination du $=$ dans **Type** :

$$\mathsf{idtoeqv}\ :\equiv\ \mathsf{transport}\ \mathsf{id}_{\mathsf{Type}}\ :\ (A = B) \to (A \simeq B)$$

Démonstration.
Il faut montrer que si $p : A = B$, alors $p_\star : A \to B$ réalise une équivalence entre A et B. C'est en fait trivial par induction sur le chemin p, car $(1_A)_\star \equiv \mathsf{id}_A$, qui est clairement une équivalence de A vers lui-même. Mais on peut également directement fournir un inverse pour $p_\star$, à savoir $p_\star^{-1}$. □

L'axiome d'univalence lui fournit un inverse, servant de règle d'introduction du $=$ dans **Type** :

$$\mathsf{ua} \ : \ (A \simeq B) \to (A = B)$$

avec les règles de calculs souhaitées :

$$\mathbf{\Pi}(f : A \simeq B), \mathsf{transport}\ \mathsf{id}_{\mathsf{Type}}\ (\mathsf{ua}\ f)\ x = f\ x$$

$$\mathbf{\Pi}(p : A = B), \mathsf{ua}\ (\mathsf{transport}\ \mathsf{id}_{\mathsf{Type}}\ p) = p$$

L'axiome d'univalence est fondamentalement important, car il permet d'unifier les notions d'équivalences et d'égalité. Il permet notamment d'introduire des chemins et rend notre modèle de calcul non trivial : il devient beaucoup plus difficile d'exhiber un modèle de la théorie des types dépendants avec axiome d'univalence (mais Vladimir Voevodsky l'a fait, avec les *ensembles simpliciaux* ! [5]). Á notre niveau, nous pouvons par exemple montrer que que cet axiome introduit des chemins non triviaux.

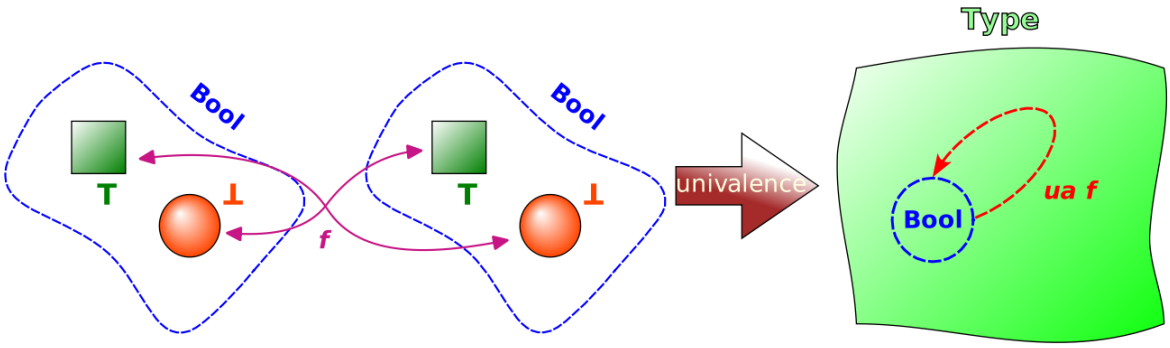
Démonstration. On considère le type **2**, et $f : \mathbf{2} \to \mathbf{2}$ la fonction définie par :

$$f :\equiv \mathbf{rec}_{\mathbf{2}}\ \mathbf{2}\ 1_2\ 0_2$$

f se comporte comme une involution qui permute les deux éléments de **2**. En particulier, il est aisé de vérifier qu'il s'agit bien d'une équivalence, et qu'elle est distincte de l'identité. Ainsi, en posant :

$$p :\equiv \mathsf{ua}\ f$$

p est une boucle sur **2** dans **Type**, et $\neg(p = 1_2)$ car $\neg(f = \mathsf{id}_2)$ et ua réalise une équivalence entre $\mathbf{2} \simeq \mathbf{2}$ et $\mathbf{2} = \mathbf{2}$. □



4 Les Types Inductifs

4.1 Les Types Inductifs classiques

4.1.1 Présentation informelle

La plupart des types que nous avons rencontrés, à l'exception de la hiérarchie des **Type** i et des **Π**-*Types* (produits dépendants), admettent une formulation en terme de type inductif. Les règles d'introduction permettent d'introduire des éléments de ce type, tandis que les règles d'élimination permettent de définir des fonctions de ces types vers d'autres types, et les règles de calcul qui vont avec déterminent le comportement de ces fonctions sur certains éléments canoniques de ces types. On veut maintenant pouvoir définir de manière générale ce que sont les types inductifs, les formations qu'on autorise et les principes d'induction (règles d'élimination) associés. On commence par une étude informelle qui se rapproche de la manière dont les types inductifs sont gérés dans le *Calcul des Constructions* (sur lequel est basé *Coq*).

Un type inductif est un nouveau type dans lequel on définit des *constructeurs* :

Inductive Foo : **Type** :=
| $\mathsf{constr}_1 : \cdots \to Foo$
| $\mathsf{constr}_2 : \cdots \to Foo$
| $\cdots$

où les constr_i jouent le rôle de constructeurs, c'est-à-dire qu'ils permettent d'introduire des éléments dans notre nouveau type Foo et vont ainsi "engendrer librement le type". Ils peuvent prendre zero argument ($\mathsf{constr}_i : Foo$), auquel cas ce sont comme des constantes ajoutées au type, ou plusieurs arguments, qui peuvent eux-même être de type Foo (voilà pourquoi on parle de types inductifs).

le type Nat On encode le type des entiers naturels. C'est le type inductif non trivial le plus simple :

Inductive **Nat** : **Type** :=
| 0 : **Nat**
| S : **Nat** → **Nat**

Sa structure reflète le fait qu'un entier naturel est soit 0, soit le successeur d'un autre entier naturel. Son principe d'induction traduit exactement la récurrence usuelle sur les entiers naturels. Soit P : **Nat** → **Type** une famille de types indexée par les entiers naturels, pour avoir un élément f : **Π**(n : **Nat**), $P\ n$ (donc une preuve de "pour tout n, $P\ n$), il suffit de donner une preuve de $P\ 0$ et de **Π**(n : **Nat**), $P\ n \to P\ (\mathsf{S}\ n)$:

$$\mathbf{ind}_{Nat} : \mathbf{\Pi}(P : \mathbf{Nat} \to \mathbf{Type}), P\ 0 \to (\mathbf{\Pi}(n : \mathbf{Nat}), P\ n \to P\ (\mathsf{S}n)) \to \mathbf{\Pi}(n : \mathbf{Nat}), P\ n$$

De manière générale et tout à fait informelle, comme un type inductif est une "partie librement engendrée" par ses constructeurs, définir une fonction sur ce type revient à se donner pour chaque constructeur une fonction vue comme le processus de calcul de l'image d'un élément donné par ce constructeur, étant donnés les arguments pris par ce constructeur ainsi que les images (déjà construites) des arguments quand ceux-ci sont du même type inductif. C'est exactement ce que dit le principe d'induction des entiers.

Voyons ce que cela donne avec le type des listes :

Inductive **List** (A : **Type**) : **Type** :=
| nil : **List** A
| cons : $A \to$ **List** $A \to$ **List** A

Définir une fonction sur les listes vers un type B se fait exactement comme par *pattern-matching* : on fournit une valeur pour la liste vide nil et donne une procédure pour construire l'image d'une liste non vide en faisant éventuellement intervenir l'image de la queue. A partir de :

$$c_{nil} : B \quad \text{et} \quad c_{cons} : A \to \mathbf{List}\ A \to B \to B$$

on peut définir la fonction souhaitée :

$$f :\equiv \mathbf{rec}_{ListA}\ B\ c_{nil}\ c_{cons} : \mathbf{List}\ A \to B$$

avec les réductions suivantes :

$$f\ \mathsf{nil} \equiv c_{nil} \quad \text{et} \quad f\ (\mathsf{cons}\ a\ l) \equiv c_{cons}\ a\ l\ (f\ l)$$

Pour l'éliminateur dépendant $\mathbf{ind}_{ListA}$, on doit fournir la fibration P : **List** $A \to$ **Type**, ainsi que :

$$c_{nil} : P\ \mathsf{nil} \quad \text{et} \quad c_{cons} : \mathbf{\Pi}(a : A)(l : \mathbf{List}\ A), P\ l \to P\ (\mathsf{cons}\ a\ l)$$

On peut se demander maintenant si tout constructeur de la forme :

$$\mathsf{constr_i} : T_1^{(i)} \to \cdots \to T_{n_i}^{(i)} \to Foo$$

est valide, *i.e.* si tout type inductif défini avec ce genre de constructeurs est valide et n'entraine pas l'inconsistence de la théorie. On remarque que si dans l'un des $T_k^{(i)}$, le type Foo apparait à gauche d'une flèche, par exemple :

$$\mathsf{constr} : (Foo \to \mathbf{Nat}) \to Foo$$

cela pose problème. Comment définir le principe de récurrence pour se constructeur ? Comme Foo est le domaine et non le co-domaine de $Foo \to \mathsf{Nat}$, la notion de "la valeur que prend f sur $s : foo \to \mathbf{Nat}$" n'a plus de sens.

On s'imposera donc que toutes les occurences de Foo appraissent à droite des flèches dans les types des arguments de chaque constructeur. Dans ce cas, la théorie est cohérente et le principe d'induction est facile à définir. Prenons les arbres dont les noeuds, étiquetés par des entiers naturels, ont un nombre dénombrable de fils (cette "infinité" d'arguments se code en fait par une famille indexée sur **Nat**) :

Inductive $Tree$: **Type** :=
| nil : $Tree$
| node : **Nat** $\to$ (**Nat** $\to$ $Tree$) $\to$ $Tree$.

Le principe de récurrence $\mathbf{rec}_{Tree}$ requiert alors un type B ainsi que :

$$c_{nil} : B \quad \text{et} \quad c_{node} : \mathbf{Nat} \to (\mathbf{Nat} \to Tree) \to (\mathbf{Nat} \to B) \to B$$

L'argument de type **Nat** $\to B$ dans c_{node} correspondant aux images de f sur les fils du noeud.

4.1.2 Une formalisation : les W-Types

Il existe une syntaxe particulière, avec un constructeur de type **W**, qui permet d'introduire les types inductifs, de même que **Π** et **Σ** introduisent respectivement les produits et sommes dépendants. En pratique, ce n'est pas ainsi que les assistants de preuve traitent les types inductifs : on préfère une syntaxe telle qu'exposée avant, qui est plus facile à gérer informatiquement et plus aisée pour le programmeur. Néanmoins, les **W**-*Types* (pour "Well Ordered Trees") permettent de bien généraliser la notion de type inductif simple.

Un type inductif se forme ainsi :

$$\mathbf{W}(x : A), P\ x : \mathbf{Type} \quad \text{avec} \quad A : \mathbf{Type} \quad \text{et} \quad P : A \to \mathbf{Type}$$

Intuitivement, A correspond aux *labels* (étiquettes) des constructeurs, et pour tout $a : A$, $P\ a$ représente l'*arité* du constructeur étiqueté par le label a, *i.e.* le "nombre" d'arguments de type $\mathbf{W}(x : A), P\ x$ qu'il prend. Le type **Nat** est relativement simple. Il a deux constructeurs : 0 : **Nat** d'arité 0 et S : **Nat** $\to$ **Nat** d'arité 1. On choisit donc $A :\equiv \mathbf{2}$ et P la fibration qui en 0_2 vaut **0** et en 1_2 **1**, *i.e.* $P :\equiv \mathbf{rec}_2$ **Type 0 1**.

Attention, cette arité ne concerne que les arguments de ce type, s'il y a des arguments supplémentaires d'autres types, il faut considérer qu'il existe une version de ce constructeur associée à chaque argument de cet autre type. Par exemple, dans le type **List** B, le constructeur cons : $B \to$ **List** $B \to$ **List** B prend comme arguments un $b : B$ (la tête) et une liste en queue. Il suffit de considérer qu'on a, pour chaque $b : B$, un constucteur $\mathsf{cons_b}$: **List** $B \to$ **List** B. À cela s'ajoute le constructeur constant nil : **List** B. **List** B se code donc ainsi :

$$\mathbf{W}(x : \mathbf{1} + B), \mathbf{rec}_{1+B}\ \mathbf{Type}\ (\mathbf{rec}_1\ \mathbf{Type}\ \mathbf{0})(\lambda(b : B) \cdot \mathbf{1})$$

Pour construire un élément dans $\mathbf{W}(x : A), P\ x$, il faut spécifier quel constructeur est utilisé et affecter les arguments de ce constructeurs. On introduit ces éléments avec :

$$\mathbf{sup} : \mathbf{\Pi}(a : A), (P\ a \to \mathbf{W}(x : A), P\ x) \to \mathbf{W}(x : A), P\ x$$

Quelques exemples :

- $0 : \mathbf{Nat}$ s'obtient avec le premier constructeur (constant) du type **Nat** (défini ici comme $\mathbf{W}$-*Type*). Fournir une fonction d'affectation $\mathbf{0} \to \mathbf{Nat}$ est simple : il s'agit du principe de récursion du faux, donc $\mathbf{rec}_0\ \mathbf{Nat}$. Ainsi $0 :\equiv \mathbf{sup}\ 0_2\ (\mathbf{rec}_0\ \mathbf{Nat})$.
- De même,
  $1 :\equiv \mathbf{sup}\ 1_2\ \lambda(x : \mathbf{1}) \cdot 0$,
  $2 :\equiv \mathbf{sup}\ 1_2\ \lambda(x : \mathbf{1}) \cdot 1$,
  ...
- Pour les listes :
  $\mathsf{nil} :\equiv \mathbf{sup}\ (\mathsf{inl}\ \star)(\mathbf{rec}_0\ \mathbf{List}\ B)$
  $\mathsf{cons}\ b\ l :\equiv \mathbf{sup}\ (\mathsf{inr}\ b)(\lambda x \cdot l)$

Le principe d'induction se définit aisément : pour définir une fonction de $\mathbf{W}(x : A), P\ x$ vers un prédicat $Q : (\mathbf{W}(x : A), P\ x) \to \mathbf{Type}$, il suffit de fournir une fonction c qui pour chaque constructeur et affectation des arguments du constructeurs, est capable de construire l'image de cet élément, en faisant intervenir les images des arguments :

$$c : \mathbf{\Pi}(a : A)(\iota : P\ a \to \mathbf{W}(x : A), P\ x), (\kappa : \mathbf{\Pi}(u : P\ a), Q\ (\iota\ u)) \to Q\ (\mathbf{sup}\ a\ \iota)$$

Alors :

$$f :\equiv \mathbf{rec}_{W(x:A),P\ x}\ Q\ c \quad : \mathbf{\Pi}(w : \mathbf{W}(x : A), P\ x), Q\ w$$

avec comme règle de calcul :

$$f\ (\mathbf{sup}\ a\ \iota) \equiv c\ a\ \iota\ (\lambda(u : P\ a) \cdot f\ (\iota\ u))$$

On a ainsi exactement le comportement attendu pour f : elle est définie par le calcul sur les éléments construits avec **sup** en procédant à des *appels récursifs* sur les éléments plus petits qui le composent.

4.1.3 Questions d'unicité

A priori, rien ne prouve que deux fonctions vérifiant les mêmes propriétés de récurrence sont égales. Mais en fait, le principe d'induction est assez expressif pour pouvoir donner une preuve d'égalité (à homotopie près).

Démonstration.
Supposons que deux fontions $f\ g : (\mathbf{W}(x : A), P\ x) \to B$ vérifient toutes les deux, pour une certaine fonction c :
$f\ (\mathbf{sup}\ a\ \iota) \equiv c\ a\ \iota\ (\lambda(u : P\ a) \cdot f\ (\iota\ u))$
$g\ (\mathbf{sup}\ a\ \iota) \equiv c\ a\ \iota\ (\lambda(u : P\ a) \cdot g\ (\iota\ u))$

On veut une preuve de :

$$Q : \mathbf{\Pi}(w : \mathbf{W}(x : A), P\ x), f\ w = g\ w$$

mais par principe d'induction (dépendant), il suffit de le vérifier sur $\mathbf{sup}\ a\ \iota$ sachant que pour tout $u : P\ a$, $f\ (\iota\ u) = g\ (\iota\ u)$. Avec extensionalité fonctionnelle, on a donc $\lambda(u : P\ a) \cdot f\ (\iota\ u) \quad = \quad \lambda(u : P\ a) \cdot g\ (\iota\ u)$, et ainsi $f\ (\mathrm{sup}\ \ a\ \iota) = g\ (\mathbf{sup}\ a\ \iota)$. On a donc prouvé Q. De nouveau par extensionalité fonctionnelle, on a $f = g$. □

Enfin, notons que les types eux-mêmes sont uniquement déterminés dès lors qu'ils vérifient le même principe d'induction. Par exemple, le type **Nat** que nous avons défini en premier, avec la syntaxe proche de *Coq*, est égal à celui défini de manière analogue mais avec les **W**-*Types*.

Démonstration.
On notera **Nat** le type des entiers naturels défini en premier et $\mathbf{Nat}^{\mathbf{W}}$ celui défini par **W**-*Type*. On définit $f : \mathbf{Nat} \to \mathbf{Nat}^{\mathbf{W}}$ récursivement en envoyant :
$0 \mapsto \mathbf{sup}\ 0_2\ (\mathbf{rec}_0\ \mathbf{Nat}^{\mathbf{W}})$
$\mathsf{S}\ n \mapsto \mathbf{sup}\ 1_2\ (\lambda x \cdot f\ n)$

puis $g : \mathbf{Nat}^{\mathbf{W}} \to \mathbf{Nat}$ par :
$\mathbf{sup}\ 0_2\ \iota \mapsto 0$
$\mathbf{sup}\ 1_2\ \iota \mapsto \mathsf{S}\ (g\ (\iota\ \star))$

Il est alors clair que $g \circ f$ (respectivement $f \circ g$) vérifie le même principe de récurrence que $\mathsf{id}_{\mathbf{Nat}}$ (respectivement $\mathsf{id}_{\mathbf{Nat}^{\mathbf{W}}}$), donc on a les égalités par la preuve précédente. On a ainsi $\mathbf{Nat} \simeq \mathbf{Nat}^{\mathbf{W}}$, et donc $\mathbf{Nat} = \mathbf{Nat}^{\mathbf{W}}$ par univalence. □

4.2 Les Types Inductifs d'Ordre Supérieur

4.2.1 Introduction aux types inductifs d'ordre supérieur

Les types inductifs d'ordre supérieur (*Higher Order Types* ou HIT en anglais) sont une extension des types inductifs classiques. En plus de constructeurs qui permettent d'introduire des points dans le nouveau type, on s'autorise maintenant des constructeurs qui ajoutent des chemins voire des chemins supérieurs dans ce type. On aura ainsi la possibilité de définir des types qui se comportent comme des "objets topologiques" bien connus, tels l'intervalle, les sphères, etc.

Contrairement aux types inductifs simples où on pouvait exprimer simplement les principes d'élimination, cela devient plus complexe pour les types inductifs d'ordre supérieur, dans la mesure où il faut spécifier le comportement de la fonction à créer non seulement sur les points du type, mais aussi sur les chemins qu'on y a ajoutés. C'est un sujet actuellement en plein développement que de donner une définition la plus générale possible des HIT (*cf* par exemple [4], [8] et [6]). Nous allons juste donner quelques idées de la manière de définir l'action d'une fonction sur des chemins et des chemins d'ordre supérieurs.

On définira les HIT avec une syntaxe analogue à celle utilisée auparavant, proche de celle de *Coq*. Attention néanmoins, il n'est pas possible de définir nativement des types inductifs d'ordre supérieur en *Coq*, il faudra recourir à des astuces, comme par exemple axiomatiser les constructeurs qu'on ne peut définir autrement. Dans notre cadre, on autorisera les constructeurs qui introduisent des points, des chemins entre des points de ce type, des chemins d'ordre 2 entre ce genre de chemins, etc. Voici des exemple de HIT que nous étudierons ensuite :

Inductive $\mathbf{I}$: **Type** :=
| 0_{I} : $\mathbf{I}$
| 1_{I} : $\mathbf{I}$
| seg : $0_{\mathsf{I}} = 1_{\mathsf{I}}$

Inductive $\mathbf{S}^1$: **Type** :=
| base_1 : S^1
| loop : $\mathsf{base}_1 = \mathsf{base}_1$

Inductive $\mathbf{S}^2$: **Type** :=
| base_2 : $\mathbf{S}^2$
| surf : $1_{\mathsf{base}_2} = 1_{\mathsf{base}_2}$ (on rappelle que 1_{base_2} signifie : idpath base_2)

Le principe d'induction doit exprimer que pour définir une fonction f d'un HIT vers un autre type B, il faut "plonger" la structure du HIT à l'intérieur de B. Il faut donc donner des images pour les points du type introduits par les constructeurs, comme on l'a fait auparavant. Mais il faut également choisir "une image" pour les chemins. Par exemple, pour l'intervalle $\mathbf{I}$, si on choisit d'envoyer $0_\mathbf{I}$ sur x et $1_\mathbf{I}$ sur y dans un certain type B, il faut également fournir un chemin $p : x = y$ dans B. La fonction $f : \mathbf{I} \to B$ qui en résultera "enverra" seg sur p via fonctorialité, *i.e.* $\mathsf{ap}\ f\ \mathsf{seg} = p$. Le chemin p impose ainsi de choisir les images x et y égales (propositionnellement) dans B, ce qui est logique puisque $\mathsf{seg} : 0_\mathbf{I} = 1_\mathbf{I}$ impose l'égalité entre $0_\mathbf{I}$ et $1_\mathbf{I}$ dans $\mathbf{I}$, et l'égalité (propositionnelle) est une relation de congruence (puisque toutes les fonctions doivent être continues !). Ces propriétés traduisent le fait que le HIT créé est "librement engendré" en tant qu'∞-groupoïde par ses constructeurs.

Le principe d'induction (donc dans le cas dépendant) est plus difficile à définir. Pour l'intervalle $\mathbf{I}$, si on envoie $0_\mathbf{I}$ sur x et $1_\mathbf{I}$ sur y dans la fibration $P : \mathbf{I} \to \mathbf{Type}$, il faut définir un "chemin dépendant" de $x : P\ 0_\mathbf{I}$ à $y : P\ 1_\mathbf{I}$ au-dessus de seg. C'est exactement ce que $\mathsf{transport}$ défini plus haut permet de faire : étant donné $\mathsf{seg} : 0_\mathbf{I} = 1_\mathbf{I}$, on envoie $x : P\ 0_\mathbf{I}$ vers $\mathsf{seg}_\star\ x : P\ 1_\mathbf{I}$, il reste alors à fournir p un chemin de type $\mathsf{seg}_\star\ x = y$, type que l'on appelle "chemins dépendants de x à y (dans la fibration p), noté : $x =^P_{\mathsf{seg}} y$. Le comportement de $f : \mathbf{\Pi}(x : \mathbf{I}), P\ x$ sur seg sera logiquement : $\mathsf{apd}\ f\ \mathsf{seg} = p$.

A chaque fois que l'on monte d'un étage dans la hiérarchie des chemins, on gagne inévitablement en complexité de définitions des règles d'élimination (surtout dans le cas dépendant).

4.2.2 L'intervalle

On vient de déterminer entièrement le type intervalle $\mathbf{I}$ avec ses règles d'élimination. C'est le HIT le plus simple à définir. Il s'agit en fait d'un chemin entre 2 points libres. Le principe d'induction des chemins nous permet dans ce genre de cas de "considérer" qu'il n'y a en fait qu'un point et que ce chemin est le chemin réflexif sur ce point. Par analogie, on aimerait donc prouver que $\mathbf{I}$ est équivalent à un point. Cela revient à montrer $\mathsf{iscontr}\ \mathbf{I}$.

Démonstration.
On choisit $1_\mathbf{I}$ comme centre de réduction. Il s'agit de montrer $P :\equiv \mathbf{\Pi}(x : \mathbf{I}), x = 1_\mathbf{I}$. Pour $x \equiv 0_\mathbf{I}$, on renvoie seg, et pour $x \equiv 1_\mathbf{I}$ le chemin réflexif $1_{1_\mathbf{I}}$. Reste à trouver pour seg une image dans $\mathsf{seg} =^P_{\mathsf{seg}} 1_{1_\mathbf{I}}$. Or les calculs élémentaires de transport sur les fibrations-chemins comme ici (que l'on n'a pas la place de développer...) donne $\mathsf{seg}_\star\ \mathsf{seg} = \mathsf{seg}^{-1} \cdot \mathsf{seg}$, il ne reste plus qu'à utiliser la proposition / homotopie $\mathsf{concat\_Vp}\ \mathsf{seg} : \mathsf{seg}^{-1} \cdot \mathsf{seg} = 1_{1_\mathbf{I}}$. On a ainsi notre preuve $f : P$, et donc $\langle 1_\mathbf{I}; f\rangle : \mathsf{iscontr}\ \mathbf{I}$. □

.

4.2.3 Le cercle S^1

On a défini le cercle $\mathbf{S}^1$ comme un point base_1 muni d'une boucle $\mathsf{loop} : \mathsf{base}_1 = \mathsf{base}_1$:

```
Inductive S¹ : Type :=
    | base₁ : S¹
    | loop : base₁ = base₁
```

Le principe de récurrence dit que pour définir $f : \mathbf{S}^1 \to B$, il suffit de fournir $b : B$ et $l : b = b$. On obtient alors f avec les proprités :

$$f\ \mathsf{base}_1 \equiv b \qquad \text{et} \qquad \mathsf{f\_comprule} : \mathsf{ap}\ f\ \mathsf{loop} = l$$

Notons qu'en général on souhaite conserver l'égalité définitionelle pour le point de base, mais les égalités sur les chemins seront souvent prises comme propositionelles. D'une part, c'est parce que il serait inélégant d'en demander une égalité par définition alors que le terme ap a été défini à l'intérieur de la théorie, et aurait pu être défini autrement. D'autre part, en *Coq* par exemple, ce genre de propriété doit être axiomatisée puisque *Coq* ne gère pas les HIT. Ainsi f_comprule est un axiome, qui ne peut exprimer une égalité qu'à homotopie près.

Pour l'élimination dans le cas dépendant, il faut fournir, outre la fibration $P : \mathbf{S}^1 \to \mathbf{Type}$ et $b : P\ \mathsf{base}_1$, un chemin dépendant $l : b =^P_{\mathsf{loop}} b$, ce qui n'est pas trivial (et parfois impossible, ce qui est rassurant puisqu'on ne voudrait pas que le cercle vérifie n'importe quel prédicat). Les règles de calcul sont alors :

$$f\ \mathsf{base}_1 \equiv b \qquad \text{et} \qquad \mathsf{f\_comprule} : \mathsf{apd}\ f\ \mathsf{loop} = l$$

Donnons un exemple de prédicat $P : \mathbf{S}^1 \to \mathbf{Type}$. Par exemple, le cercle $\mathbf{S}^1$ est-il contractile ? La définition de iscontr semble dire que oui, puisque pour tout point du cercle, il semble raisonnable de dire qu'il est relié à base_1. Sauf que par essence la fonction hypothétique f qui à $x : \mathbf{S}^1$ renvoie un chemin $x = \mathsf{base}_1$ doit être continue en x, autrement dit le choix des chemins qui rétractent le cercle sur un point doit se faire *continûment*, ce qui bien évidemment est impossible en topologie classique. Voyons donc où la preuve de iscontr $\mathbf{S}^1$ coince :

Démonstration. Pour définir une fonction $f :$ iscontr $\mathbf{S}^1$, il faut d'abord une preuve $p :$ iscontr $\mathbf{S}^1 \equiv \mathsf{base}_1 = \mathsf{base}_1$. Mais ensuite, on doit obtenir un terme dans $p =^{\mathsf{iscontr}\ \mathbf{S}^1}_{\mathsf{loop}} p$. Or toujours par calcul de transports dans les fibrations chemins, $\mathsf{loop}_\star\ p = \mathsf{loop}^{-1} \centerdot p$, donc *in fine* cela revient à montrer $\mathsf{loop} = 1_{\mathsf{base}_1}$. Or nous allons voir pourquoi c'est faux. □

Il semble a priori raisonnable de considérer que loop n'est pas le chemin réflexif, autrement on n'aurait plus vraiment un cercle mais juste un point. C'est justement toute l'expressivité du principe d'élimination qui garantit ce fait. En effet, le principe de récurrence dit qu'on peut définir une fonction $f : \mathbf{S}^1 \to B$ juste en donnant l'image $b : B$ et un chemin $l : b = b$. Alors ap f loop $= l$. Or nous avons vu avec l'axiome d'univalence qu'il existe des boucles non triviales (avec par exemple $B :\equiv \mathbf{Type}$, $b :\equiv \mathbf{2}$ et $l : \mathbf{2} = \mathbf{2}$ comme étant le chemin associé à l'équivalence qui permute les deux booléens). Ainsi, loop est non triviale, sinon on aurait $l =$ ap f loop $= 1_b$.

En somme, les principes d'élimnation imposent d'une part de respecter la structure du type inductif lorsqu'on l'envoie dans un autre type (de même que par exemple une application linéaire envoie deux vecteurs proportionnels vers deux vecteurs-images proportionnels). Mais d'autre part, la liberté dans le choix des images, sous réserve de respecter ces conditions, impose que le type est "le plus général possible" qui possède cette structure : ici, la boucle doit être générale, et ne peut être le cas du chemin réflexif, qui aboutirait à une contradiction en choisissant comme image une boucle non triviale. Par analogie, ce serait comme définir la notion de base dans un espace vectoriel par un "principe d'induction" qui dirait que pour définir une application linéaire de cet espace vectoriel vers un autre, il suffit de choisir des images pour les vecteurs de la base : l'aspect *génératif* est donné par le fait qu'on caractérise entièrement l'application par sa donnée sur la base, et l'aspect *libre* résulte du fait qu'on peut envoyer les vecteurs de la base vers n'importe quelles images, qui peuvent par exemple former une famille libre, ce qui implique la liberté de notre base d'origine.

4.2.4 Sphères et Suspensions

Il est possible de généraliser la notion de sphère en dimension n. En dimension 2, cela donne :

Inductive $\mathbf{S}^2$: **Type** :=
| $\mathsf{base}_2 : \mathbf{S}^2$
| $\mathsf{surf} : 1_{\mathsf{base}_2} = 1_{\mathsf{base}_2}$

En revanche, il est plus difficile de donner un énoncé correct des principes d'élimination, car il faut pour cela définir des notions de chemins et de transport en dimension quelconque. Pour la dimension 2, le principe de récurrence reste assez simple. En effet, si $x\ y : A$, $p : x = y$ et $f : A \to B$, on a $\mathsf{ap}\ f\ p : f\ x = f\ y$: c'est un chemin en dimension 1 qui traduit la congruence de l'égalité pour les fonctions (donc l'action de f sur les chemins d'ordre 1). On peut de même définir l'action de f sur les chemins d'ordre 2, avec $x\ y : A$, $p\ q : x = y$ et $r : p = q$:

$$\mathsf{ap2}\ f\ r :\equiv \mathsf{ap}\ (\mathsf{ap}\ f)\ r : \mathsf{ap}\ f\ p = \mathsf{ap}\ f\ q$$

Le principe d'induction est plus compliqué et nécessite d'introduire des chemins dépendants en dimension 2. (Pour l'étude de la sphère d'ordre 2, *cf* mon rapport de stage). Généraliser à une dimension arbitraire n, c'est-à-dire écrire des fonctions dont le premier argument n : **Nat** représente la dimension dans laquelle on souhaite travailler, est encore plus difficile, car on se heurte aux problèmes d'expressivité du système dans lequel on se place. Ceci est traité dans mon rapport de stage également.

Dans la pratique, pour faire des preuves plus simplement sur les sphères d'ordre n, on utilise une autre définition, souvent plus maniable, de la sphère, et ce grâce à la notion de *suspension*. La suspension est une opéaration topologique consistant à faire d'une variété de dimension n une nouvelle variété de dimension $n+1$. Pour cela, on prend la variété V que l'on "étire" suivant la $(n+1)^e$ dimension, *i.e.* $V \times [0,1]$, et on identifie tous les points de $V \times \{0\}$ ainsi que ceux de $V \times \{1\}$.

En HoTT, on définit la suspension d'un type A comme deux points N et S (pour *Nord* et *Sud*), entre lesquels il existe un chemin pour chaque $x : A$ (comme si on traçait un méridien de N à S passant par x pour chaque $x : A$) :

```
Inductive Susp (A : Type) : Type :=
  | N : Susp A
  | S : Susp A
  | merid : A → N = S
```

Le principe de récurrence dit que pour définir $f : \mathsf{Susp}\ A \to B$, il suffit de spécifier des images $n\ s : B$ ainsi qu'une fonction $m : A \to n = s$. On obtient alors une fonction f avec les propriétés de calcul :

$$f\ \mathsf{N} \equiv n \quad f\ \mathsf{S} \equiv s \quad \mathsf{f\_comprule} : \mathbf{\Pi}(x : A), \mathsf{ap}\ f\ (\mathsf{merid}\ x) = m\ x$$

Pour le principe d'induction, on remplace juste comme d'habitude l'égalité par l'égalité dépendante dans la fibration indexée par le méridien, *i.e.* :

$$n : P\ \mathsf{N} \quad s : P\ \mathsf{S} \qquad\qquad m : \mathbf{\Pi}(x : A), n =^{P}_{\mathsf{merid}\ x} s$$
$$f\ \mathsf{N} \equiv n \quad f\ \mathsf{S} \equiv s \quad \mathsf{f\_comprule} : \mathbf{\Pi}(x : A), \mathsf{apd}\ f\ (\mathsf{merid}\ x) = m\ x$$

On peut alors définir les sphères comme des suspensions itérées. Les mathématiciens considèrent usuellement que la sphère de dimension 0 est une paire de deux points, que nous représentons par le type **2**. À quoi peut bien ressembler la suspension de **2** ? Il s'agit d'une paire de points N et S reliés entre eux par deux chemins, l'un "passant" par 0_2 et l'autre par 1_2.

On peut montrer que $\mathsf{Susp}\ \mathbf{2} \simeq \mathbf{S}^1$. Il faudra donc définir des fonctions $f : \mathsf{Susp}\ \mathbf{2} \to \mathbf{S}^1$ et $g : \mathbf{S}^1 \to \mathsf{Susp}\ \mathbf{2}$ dont les composées dans les deux sens soient homotopes à l'identité. Pour les images des points, nous n'avons pas beaucoup le choix : f enverra N et S sur base_1, et g enverra base_1 sur N disons. La difficulté réside dans la façon de se faire correspondre les chemins. Intuitivement, la "boucle" dans $\mathsf{Susp}\ \mathbf{2}$ est $(\mathsf{merid}\ 0_2) \cdot (\mathsf{merid}\ 1_2)^{-1}$: elle doit donc être envoyée sur loop. On peut donc par exemple envoyer $\mathsf{merid}\ 0_2$ sur loop et $\mathsf{merid}\ 1_2$ sur 1_{base_1}, pour qu'ensuite g renvoie loop sur $(\mathsf{merid}\ 0_2) \cdot (\mathsf{merid}\ 1_2)^{-1}$. Ainsi :

$f :\equiv \mathbf{rec}_{Susp\ 2}\ \mathbf{S}^1\ \mathsf{base}_1\ \mathsf{base}_1\ (\mathbf{rec}_2\ (\mathsf{base}_1 = \mathsf{base}_1)\ \mathsf{loop}\ 1_{\mathsf{base}_1})$
$g :\equiv \mathbf{rec}_{S^1}\ (\mathsf{Susp}\ \mathbf{2})\ \mathsf{N}\ ((\mathsf{merid}\ 0_2)\ \centerdot\ (\mathsf{merid}\ 1_2)^{-1})$

Démonstration.
On veut montrer $P :\equiv \mathbf{\Pi}(x : \mathsf{Susp}\ \mathbf{2}), g\ (f\ x) = x$. Pour N et S, on renvoie respectivement les chemins 1_{N} et $\mathsf{merid}\ 1_2$. Il reste à montrer que pour tout $x : \mathbf{2}$ (donc en fait $x \equiv 0_2$ et $x \equiv 1_2$) , $1_{\mathsf{N}} =^{P}_{\mathsf{merid}\ x} \mathsf{merid}\ 1_2$. Or $(\mathsf{merid}\ x)_\star\ 1_{\mathsf{N}} = (\mathsf{ap}\ (g \circ f)\ (\mathsf{merid}\ x))^{-1}\ \centerdot\ 1_{\mathsf{N}}\ \centerdot\ \mathsf{merid}\ x = (\mathsf{ap}\ (g \circ f)\ (\mathsf{merid}\ x))^{-1}\ \centerdot\ \mathsf{merid}\ x$. Comme $\mathsf{ap}\ g\ (\mathsf{ap}\ f\ (\mathsf{merid}\ 0_2)) = (\mathsf{merid}\ 0_2)\ \centerdot\ (\mathsf{merid}\ 1_2)^{-1}$ et $\mathsf{ap}\ g\ (\mathsf{ap}\ f\ (\mathsf{merid}\ 1_2)) = 1_{\mathsf{N}}$, c'est clair.

Pour montrer dans l'autre sens $Q :\equiv \mathbf{\Pi}(y : \mathsf{Susp}\ \mathbf{2}), f\ (g\ y) = y$, pour $y \equiv \mathsf{base}_1$ on renvoie 1_{base_1}, et il reste à prouver que $1_{\mathsf{base}_1} =^{Q}_{\mathsf{loop}} 1_{\mathsf{base}_1}$. Or $\mathsf{loop}_\star\ 1_{\mathsf{base}_1} = (\mathsf{ap}\ (f \circ g)\ \mathsf{loop})^{-1}\ \centerdot\ 1_{\mathsf{base}_1}\ \centerdot\ \mathsf{loop} = (\mathsf{ap}\ f\ ((\mathsf{merid}\ 0_2)\ \centerdot\ (\mathsf{merid}\ 1_2)^{-1}))^{-1}\ \centerdot \mathsf{loop} = \mathsf{loop}^{-1}\ \centerdot\ \mathsf{loop} = 1_{\mathsf{base}_1}$. Donc c'est prouvé. □

On peut donc définir de manière récursive sur $n : \mathbf{Nat}$:

$\mathsf{Fixpoint}\ \mathbf{S}'\ (n : \mathbf{Nat}) : \mathbf{Type} :=$
$\quad | \ 0 \mapsto \mathbf{2}$
$\quad | \ \mathsf{S}\ p \mapsto \mathsf{Susp}\ (\mathbf{S}'\ p)$

Une importante partie de mon stage a porté sur les preuves que cette définition des sphères en dimension n quelconque équivaut à une définition plus dans le goût de ce qu'on a présenté avant avec $\mathbf{S}^1$ et $\mathbf{S}^2$. Cela demande un travail préliminaire de définitions assez long, qui est détaillé dans le rapport du stage.

4.2.5 Et ensuite ?

Les possibilités ne s'arrêtent pas là ! On peut définir bien d'autres variétés, comme le tore, le cône, le plan projectif, la bouteille de Klein, et toutes sortes de variétés obtenues par recollement, quotient, etc. Néanmoins, les principes d'inductions deviennent vite très lourds et nécessitent de nombreuses nouvelles définitions. C'est pourquoi ils sont souvent mentionnés et décrits de manière informelle, mais rarement décrits rigoureusement et encore moins implémentés sur un assistant de preuve.

Outre la possibilité de définir ces variétés en HoTT, on peut prouver des propriétés homotopiques de ces variétés grâce à nos représentations dans HoTT. L'exemple le plus fréquent et assez marquant est la preuve que $\pi_1(S^1) = \mathbb{Z}$, où $\pi_1(S^1)$ désigne le *groupe fondamental* du cercle. La preuve en HoTT se fait avec le type $\mathbf{S}^1$ que nous avons défini !

Ce n'est qu'un exemple des remarquables perspectives offertes par HoTT. Cette mise en évidence des liens très profonds entre théorie de l'homotopie et théorie des types n'a pas encore livré tous ses secrets !

Références

[1] Henk Barendregt. *The Lambda-Calculus*. Elsevier Science Publishing Company, 1984.

[2] Yves Bertot et Pierre Castéran. *Coq'Art : The Calculus of Inductive Constructions*. Springer, 2004. url : `http://www.labri.fr/perso/casteran/CoqArt/coqartF.pdf`.

[3] *Homotopy Type Theory*. url : `http://homotopytypetheory.org`.

[4] *Homotopy Type Theory, Univalent Foundations of Mathematics*. Institute for Advanced Study, 2013. url : `http://hottheory.files.wordpress.com/2013/03/hott-online-207-g21ac918.pdf`.

[5] Chris Kapulkin, Peter LeFanu Lumsdaine et Vladimir Voevodsky. *The simplicial model of univalent foundations*. 2012. url : `http://arxiv.org/abs/1211.2851`.

[6] Peter LeFanu Lumsdaine. *Higher Inductive Types : a tour of the menagerie*. 2011. url : http://homotopytypetheory.org/2011/04/24/higher-inductive-types-a-tour-of-the-menagerie/.

[7] Per Martin Löf. *Intuitionistic type theory*. T. 1. Studies in Proof Theory. Bibliopolis, 1984.

[8] Egbert Rijke. *Homotopy Type Theory*. 2012. url : http://hottheory.files.wordpress.com/2012/08/hott2.pdf.

[9] Wikipedia. *Correspondance de Curry-Howard*. url : http://fr.wikipedia.org/wiki/Curry-Howard.

[10] Wikipedia. *Dependent Types*. url : http://en.wikipedia.org/wiki/Dependent_type.