Vérification formelle de la construction du groupe fondamental à l'aide de l'assistant de preuve LEAN

Naomi Jacquet

TER encadré par Frédéric Le Roux

9 juin 2020

Table des matières

Première partie

Introduction

1 Lean

Lean est un assistant de preuve basé sur la théorie des types et sur le lambda-calcul. Lean est capable de vérifier la véracité d'un énoncé mathématique, pour peu qu'il soit correctement écrit dans le langage de la théorie des types. La puissance de cet assistant a fait qu'une bibliothèque assez conséquente de résultats, définitions et démonstrations a été créée par la communauté[1]. Bien qu'elle contienne de nombreux résultats de topologie et d'analyse réelle, cette bibliothèque (nommée mathlib) ne contient aucune théorie de topologie algébrique. Le but de ce TER est de définir la notion de groupe fondamental d'un espace topologique au sein de Lean et de démontrer tout les résultats qui permettent d'y parvenir à partir des notions déjà présentes dans mathlib.

Le contenu mathématique suit les preuves classiques sur la définition du groupe fondamental, on pourra par exemple les voir dans [Ler17, Ite18], qui explicite quasi-toutes les homotopies nécessaires à la construction du π_1.
Les fichiers de ce TER ainsi que les instructions pour accéder aux preuves sont disponible à cette adresse : `https://github.com/Naomij1/lean-homotopie`

2 Types

La fondation de lean ne repose pas sur les ensemble — contrairement à la fondation la plus populaire des mathématiques — mais sur des types, qui ressemblent à ceux des langages de programmation. On peut voir intuitivement qu'une telle notion si centrale en informatique est plus simple à utiliser dans un assistant de preuve. Ainsi chaque objet mathématique possède un type, qu'on peut toujours remonter, en prenant l'opération « type du type », à "`Type`" qui nous sert d'« univers ». Le typage est noté `:` dans Lean, et on peut donner comme exemple $(0 : \mathbb{N})$ et $(\mathbb{N} : \texttt{Type})$.

3 Lambda calcul

Le lambda calcul est un modèle abstrait qui va nous permettre de définir des fonctions. On dispose de deux opérations : l'abstraction et l'application. La première nous définit un élément du type $A \to B$, le type des fonctions de A vers B, et correspond à $f : x \to y$, elle est notée `λ x, y`. On a également la possibilité d'appliquer une fonction f à une variable x, qui nous donne $f(x)$ et est noté `f x`.

[1] On peut trouver la documentation ici : `https://leanprover-community.github.io/mathlib_docs`

4 La preuve

La force de la théorie des types comme fondation des assistants de preuve repose sur l'isomorphisme de Curry-Howard : à la fois les objets mathématiques et les propositions sur ces objets sont des types ; et les règles du lambda calcul correspondent à celles de la logique. Ainsi à toute proposition on peut associer une expression typée ; et à chaque démonstration formelle on peut associer un algorithme.

Donc, l'interprétation des formules en tant que type fait qu'une démonstration (un algorithme) correspond à la création d'un élément du type de la proposition qu'on souhaite démontrer. Dans Lean, les propositions sont de type `Prop`, et démontrer $(A$: `Prop`$)$ consiste à exhiber x : `A`. En guise d'exemple trivial :

Exemple 4.1. Soit A une proposition (i.e. un élément du type `Prop`), alors $A \implies A$

Démonstration. Démontrer $A \implies A$, par l'isomorphisme de Curry-Howard, revient à construire un terme de type $A \to A$ (la théorie des types identifie l'implication entre A et B avec le type des fonctions de A vers B).
On considère $\lambda(x : A),\ x$, qui est bien de type $A \to A$. On peut écrire la preuve dans Lean comme suit :

```
example  (A: Prop) : A → A := λ x, x
```

□

Il existe deux manières d'écrire une démonstration dans Lean : par utilisation directe du λ-calcul (comme la preuve ci-dessus), ou par utilisation de tactiques. Par exemple, pour la règle d'application du λ-calcul typé, qui correspond à l'élimination de l'implication :

$$\frac{\Gamma \vdash f : \alpha \to \beta \qquad \Gamma \vdash x : \alpha}{\Gamma \vdash f(x) : \beta}$$

Comme pour la déduction naturelle, on peut lire cette règle de haut en bas ou de bas en haut (voir [Ned14] pour plus de détails).

1. De haut en bas, cela correspond au fait qu'étant donné $f : \alpha \to \beta$ et x de type α, l'application de f en α nous donne un élément de type β. On écrira dans Lean le code f x
2. De bas en haut, cela correspond au fait que pour expliciter un terme de type β, on peut exhiber un élément de type α, et une fonction de type $\alpha \to \beta$. En Lean, cela correspond à l'utilisation de la tactique `apply f`, qui passe l'objectif de démonstration de β à α

L'utilisation des tactiques (identifiables par les mots-clefs `begin` et `end`) font que l'objectif de ce qui est à démontrer est affiché par l'IDE, et que l'on manipule essentiellement cet objectif afin de revenir à nos hypothèses. Les tactiques utilisées dans les fichiers sont : `apply, split, simp, intro, congr, rw, split_ifs, exfalso`. Présentons les brièvement :

- `exact` est une variante de `apply` qui clôt un objectif.
- `split` est l'introduction de la conjonction. Cette tactique permet de transformer une preuve de $p \wedge q$ en une preuve de p et en une preuve de q

- `intro` est une règle globale d'introduction. Elle permet notamment d'introduire l'application et les quantificateurs existentiels.
- `rw` réécrit une hypothèse en paramètre dans une égalité.
- `simp` effectue des simplifications et des réécritures que Lean possède dans sa bibliothèque, elle permet de rendre un objectif plus lisible, de démontrer des résultats très simple ou de réécrire légèrement ce qu'on cherche à démontrer.
- `congr` transforme une preuve de $f(x) = f(y)$ en une preuve de $x = y$. Cette tactique est irréversible, mais très utile.
- `split_ifs` nous permet de traiter l'égalité dans les fonctions conditionnelles en traitant tous les cas possibles.
- `exfalso` correspond au principe d'explosion. Si on a démontré `false`, on peut en déduire ce qu'on veut.
- `linarith`, utilisée en combinaison d'`exfalso` cherche une contradiction entre les hypothèses sous forme d'inégalités et la bibliothèque de résultat sur les inégalités de Lean. Cette tactique est puissante, mais d'utilisation limitée.

5 Typeclasses

Le concept de typeclass est hérité de Haskell, qui permet d'indiquer que certains types fonctionnent d'une manière commune. En Lean cela correspond à ce que certaines familles de types possèdent des propriétés communes, et évite de redémontrer des résultats vrais en toute généralité : par exemple, si un type possède la typeclass `comm_semiring` (demi-anneau commutatif), la formule de Newton (`add_pow`) sera vraie sans avoir à la redémontrer.

Dans les fichiers nous utiliserons surtout explicitement la typeclass `topological_space`, mais implicitement beaucoup d'autres, e.g. le concept d'anneau, de corps, de groupe, etc.
Chaque typeclass peut être instanciée pour un type particulier afin d'obtenir les propriétés définies précédemment, par exemple, le typeclass `pointed` que nous définirons ultérieurement est instancié pour $\mathbb{R}$ par le code suivant.

```
instance : pointed ℝ := pointed.mk 0
```

Cela signifie que par la suite, quand nous parlerons de $\mathbb{R}$, nous parlerons de l'espace pointé $(\mathbb{R}, 0)$. Cette opération est appelée une instanciation, et tout l'objectif du TER consiste à instancier la typeclass `group` à l'ensemble des lacets à homotopie près. On pourra se référer à [AdMK20] pour plus de détails sur les typeclasses.

6 Sous-types

Le concept de sous-type est essentiel au TER. Non seulement, la notion d'ensemble dans mathlib est codé par des sous-types, mais presque tous les objets que nous manipulerons sont des éléments de sous-types. Un sous-type est un type définit par un prédicat [mC20, AdMK20], il « correspond » en théorie des ensembles à l'ensemble

$$\{x \in E, P(x)\}$$

Dans la syntaxe de Lean, cela donne :

```
inductive subtype { α : Type u} (p : α -> Prop) -- Définition formelle
| mk : ∀ x : α, p x -> subtype

{x : α // p α} -- notation abrégée, basée sur celle des ensembles
```

Un élément d'un tel sous-type n'aura pas pour type α, mais bien `{x : α // p α}`. Cela signifie donc qu'on ne peut pas procéder de la même manière que si on manipulait des objets de α dans les preuves et les définitions. Un sous-type contient deux champs :

1. `val`, qui est la valeur intrinsèque de la variable dans le type ambiant. `x.val` nous permet ainsi d'obtenir la valeur de x.
2. `prop`, qui est la propriété que doit satisfaire un élément du sous-type.

Pour définir un membre d'un sous-type, on doit se donner sa valeur (le `x : α` dans la définition de `mk`), ainsi qu'une preuve de p α. Lean possède une notation propre qui permet d'abréger la construction d'un élément d'un soustype :

```
def y : {x : α // p α}  := ⟨ y_val, preuve_de_p_y ⟩
```

Lean possède un système pour passer d'un sous-type vers le type ambiant appelé coercion, qu'on instancie par la typeclass `has_coe` et qu'on utilisera implicitement dans le reste des fichiers.

Deuxième partie
Guide des fichiers

7 Définitions

misc.lean

Le fichier `misc.lean` contient essentiellement la définition de $I = [0, 1]$ et ses propriétés élémentaires. I est de type `set`, c'est donc un sous-type de $\mathbb{R}$ dont les éléments ont la propriété d'être entre 0 et 1.
Par exemple pour montrer que 0 appartient à I, nous procédons comme suit :

```
instance : has_zero I := ⟨⟨(0:ℝ), and.intro (le_refl 0) (zero_le_one)⟩⟩
```

Cela signifie que nous instancions la typeclass `has_zero` en le 0 de $\mathbb{R}$. Cela revient à montrer que $0 \in I$. Détaillons brièvement la preuve : notons que `le_refl 0` a pour type $0 \leq 0$ et que `zero_le_one` a pour type $0 \leq 1$. L'introduction de la conjonction nous donne finalement une preuve de $0 \leq 0 \ \wedge \ 0 \leq 1$, ce qui est bien la propriété cherchée.

Ce fichier contient également un certain nombre de lemmes simples pas forcément démontrés mais qui évitent de rester bloquer dans certaines preuves des fichier principaux sur la démonstration de résultats triviaux.

homotopy.lean - définitions

Nous définissons formellement un lacet sur un espace topologique pointé (X, x_0) à l'aide des sous types comme suit :

```
/-- Lacet sur un type X -/
def loop := {f: I → X // continuous f ∧ f(0) = f(1) ∧ f(0) = point X}
```

Cette construction indique que nous voyons loop X comme le sous-type des fonctions f de I vers X, tel que f est continue, $f(0) = f(1)$ et $f(0) = x_0$. La propriété `continuous` est donnée par mathlib, et la possibilité d'écrire $f(0)$ et $f(1)$ nous provient de l'instanciation précédente des typesclasses `has_zero` et `has_one`.

De même nous définissons la relation d'homotopie de lacets sur X :

```
/-- Homotopie de lacets -/
def loop_homotopy (f : loop X) (g : loop X) : Prop := ∃(H : I × I → X),
    (∀ t, H(0,t) = f.val(t) ∧  H(1,t)=g.val(t)) ∧ (continuous H)
```

Cette définition décrit l'existence d'une homotopie entre deux lacets f et g, c'est à dire une fonction continue $H : I \times I \to X$ telle que $H(0, -) = f$ et $H(1, -) = g$. On notera que la notation f.val est nécessaire car f représente un objet de loop X et non une fonction.

8 Homotopie

La première partie du TER consiste à formaliser la preuve que l'homotopie forme une relation d'équivalence sur les lacets. Lean inclue les types quotients, qui font qu'une fois définie une relation d'équivalence, l'espace quotient des lacets à homotopie près s'obtient immédiatement.
Pour ceci, Lean possède des outils qui font partie de la bibliothèque standard, à savoir les propositions reflexivity, transitivity et symmetry. Toutes les preuves qui suivent dans ce fichier sont séparées en deux : d'une part on montre que l'homotopie vaut bien ce qu'on lui demande en $H(1, -)$ et $H(0, -)$, et d'autre part, on montre que l'homotopie est continue.

8.1 La refléxivité

C'est la preuve la plus simple. Elle est basée sur l'homotopie constante

$$H(s,t) = f(s)$$

La définition d'un lacet comme un soustype fait qu'on doit utiliser f.val dans la définition qui s'écrit : `H : I×I → X := λ x, f.val (x.2)`. On notera aussi x.2 qui correspond à la projection sur le deuxième élément de x qui appartient au produit $I \times I$.

Valeur aux extrémités : La tactique simp se charge de tous les détails.

Continuité de l'homotopie : Les trois ingrédients nécessaires pour montrer la continuité sont la continuité de la projection (qui découle de la construction de la topologie produit), la continuité de f et la continuité de la composition de deux fonctions continues. Toutes ces preuves sont déjà dans mathlib, et on a seulement besoin d'agencer ces éléments avec apply pour obtenir une preuve.

8.2 La symétrie

Cette preuve est un peu plus tactique. Étant donnée une homotopie H_1 entre f et g, on définit l'homotopie suivante entre g et f :

$$H(s,t) = H_1(1-s,t)$$

On retrouve les mêmes éléments de définition qu'on avait eu avec le chemin inverse : il faut montrer que $s \in [0,1] \implies (1-s) \in [0,1]$, et c'est précisément un lemme dans `misc.lean`.

Valeur aux extrémités : Ici simp ne suffit plus à démontrer le résultat souhaité. On reste bloqués sur `H ((expression compliquée, _ ), (0, t).snd)) = g.val t` Pour venir à bout de cet objectif, on emploie le lemme ad-hoc suivant :

```
lemma coe_of_0 : (0 : I).val = (0 : ℝ ) := rfl
```

Avec ce résultat, l'inférence de type de Lean semble fonctionner, et on peut conclure pour la valeur en 0 avec `rw`. Similairement le lemme `one_minus_one_coe` permet à Lean de simplifier l'expression compliquée dans le premier argument de H, qui est un sous-type.

Continuité de l'homotopie : La preuve de la continuité se résume à un long enchaînement des blocs de base déjà prouvés dans mathlib avec la tactique apply. Ce point est d'ailleurs remarquable et particulièrement pédagogique : une fois démontré des résultats élémentaires sur la continuité ($f+g$ continue si f et g continues, ...), la continuité d'un grand nombre de fonctions vient immédiatement.

8.3 La transitivité

En notant H_1 une homotopie entre f et g, et H_2 une homotopie entre g et h, l'homotopie nécessaire à démontrer le résultat est la suivante :

$$H(s,t) = \begin{cases} H_1(2t, s) & \text{si } t \leq 0,5 \\ H_2(2t-1, s) & \text{sinon} \end{cases}$$

On voit qu'on a besoin de découper l'intervalle en deux, donc d'utiliser une expression conditionelle qui s'écrit dans Lean `ite`, et qu'il faudra dire dans quelle partie de l'intervalle on se situe, ce qui sera fait à travers la tactique `split_ifs`.

Elle fonctionne comme suit pour les égalités : étant donné une proposition du type "(*si* condition *alors* x *sinon* y) = z", elle ouvre deux nouveaux objectifs

- Condition $\implies x = z$
- ¬Condition $\implies y = z$

Valeur aux extrémités : Ici, les conditions correspondent à « une extrémité appartient à une partie de l'intervalle ». Ainsi, seul l'un des deux objectifs partira avec des prémisses vrai, et l'autre sera démontré en réfutant la prémisse.

Par exemple, pour montrer que $H(0,t) = f(t)$, la tactique `split_ifs` fait une disjonction de cas selon si $0 \leq 1/2$ ou $0 > 1/2$.
Dans le premier cas, on obtient l'égalité sur la première expression de H. On la montre en utilisant `rw <-` qui réécrit « dans le sens inverse », qui nous donne une égalité entre deux applications de H_1, qu'on peut traiter à l'aide de la tactique `congr`, qui la transforme en une égalité sur les arguments.
Dans le second cas, l'hypothèse $1/2 < 0$ est absurde et `exfalso` transforme l'objectif en faux, qui est précisément ce que démontre $1/2 < 0$

Continuité de l'homotopie : En plus des propriétés vues précédemment, on a besoin de `continuous_if` qui nous permet de prouver la continuité d'une application définie par morceaux, en démontrant la continuité dans les deux parties de l'intervalle et l'égalité des deux définitions à la frontière. On utilise un lemme démontré partiellement dans `misc.lean` pour l'expression de la frontière. On peut se référer à la conclusion pour les raisons de l'inachèvement de cette preuve.

8.4 L'espace des classes d'homotopie

On finit ce fichier en définissant l'ensemble des lacets quotienté par la relation d'homotopie. On voit ici toute la puissance de Lean dans certains cas : le seul fait d'avoir démontré que cette relation est transitive, réflexive et symétrique suffit à pouvoir définir un type quotient. Cet espace est donné par `homotopy_classes` et on va lui donner une structure de groupe dans le fichier suivant.

En pratique, cela correspond au lignes suivantes :

```
/-- Sétoïde (X, homotopies de X) -/
definition homotopy.setoid : setoid (loop X) :=
    { r := loop_homotopy X, iseqv := loop_homotopy_equiv X}
```

```
/-- Ensemble des classes d'homotopie -/
definition homotopy_classes := quotient (homotopy.setoid X)
```

On notera que pour faire un quotient dans Lean, nous sommes obligés de définir un sétoïde, c'est à dire un type muni d'une relation d'équivalence. La fonction quotient suffit pour définir l'espace quotient. On finit le fichier par deux petites définitions qui ont pour but d'alléger les notations.

9 Le groupe fondamental

9.1 fundamentalgroup.lean

Ce fichier regroupe la définition du groupe fondamental à proprement parlé, ainsi que tous les lemmes nécessaires pour le définir. Ces propriétés et définitions à remplir sont liées à la manière qu'à mathlib d'implémenter la notion de groupe parmi de nombreuses définitions équivalentes, et donc de ce que nous avons à faire pour instancier le typeclass group à notre type de classes d'homotopie. Dans le fichier, cela correspond aux lignes suivantes vers la fin du fichier :

```
/- Le groupe fondamental (à remplir) -/
noncomputable instance : group (homotopy_classes X) :=
{ mul := homotopy.comp X, -- loi de composition interne
  mul_assoc := by sorry, -- associativité
  one := [const_x₀ X | X], -- élément neutre
  one_mul := const_comp X, -- l'élément neutre est neutre à gauche
  mul_one := comp_const X, -- l'élément neutre est neutre à droite
  inv := homotopy.inv X, -- inverse
  mul_left_inv := by sorry } -- l'inverse donne le neutre à gauche
```

Le plan général de ce fichier est donc :

1. Montrer que la composition passe au quotient
2. Montrer que la composition est associative
3. Définir un élément neutre, le lacet constant égal à x_0, qu'on note c_0
4. Montrer que $[f \cdot c_0] = [f]$
5. Montrer que $[c_0 \cdot f] = [f]$
6. Montrer que l'inverse passe au quotient (une étape souvent oubliée des textes, même les plus verbeux)
7. Montrer que $[\bar{f} \cdot f] = [c_0]$

9.2 La composition passe au quotient

Cette preuve a été omise. On peut se référer à la conclusion pour en avoir la raison.

9.3 L'inverse passe au quotient

Comme on a déjà défini l'inverse dans le fichier homotopy.lean, il faut montrer que celui ci est bien compatible avec la réduction à homotopie près. Ainsi étant donnée deux application f et g et une homotopie H_1 entre elles, on construit :

$$H(s,t) = H_1(s, 1-t)$$

La preuve est assez semblable aux précédentes, mais on peut remarquer trois choses :

Premièrement l'équivalence $f_1 \approx f_2$ peut être explicitée par la tactique `cases` qui va introduire l'homotopie H_1 et les propriétés qu'elle vérifie. Cette tactique « détruit » la relation d'équivalence et nous permet d'écrire une preuve de continuité / égalité à la frontière comme précédemment.

Ensuite, on utilise la propriété `quotient.sound`. Elle est définie comme suit dans la bibliothèque standard de Lean.

```
constant sound : ∀ α {r : α → α → Prop} {a b : α},
  r a b → quot.mk r a = quot.mk r b -- quot.mk est la réduction mod r
```

Une telle propriété est vérifiée pour tous les quotients, ainsi, elle nous permet de passer de la démonstration d'une égalité entre deux classes vers la démonstration d'une équivalence entre deux représentants quelconques. On introduit H comme définie précédemment ; et la tactique `use` indique à Lean qu'on souhaite poursuivre la démonstration avec cette définition de H.

Il est à noter que la preuve de l'appartenance de $1-t$ à l'intervalle I change le typage des éléments dans Lean. C'est un point important, car on doit s'assurer d'avoir des preuves d'appartenance égales pour des éléments égaux. Cette difficulté est la cause d'un manque de démonstration dans la suite du fichier (voir §Preuves omises dans la conclusion).

Finalement, la définition `inv` fait appel à la propriété `quot.lift`, qui « relève » la fonction vers l'espace quotient.

9.4 Élément neutre

On définit ce qui va nous servir d'élément neutre pour le groupe fondamental, à savoir le lacet constant égal à x_0 :

```
/-- La boucle constante égale à point X -/
def const_x : loop X :=  ⟨λ x, point X,
    by {split, assume h,  exact continuous_const h, split, refl, refl}⟩
```

Comme pour tous les sous-types, la définition est découpée en la construction de l'objet et la preuve qu'il appartient bien au sous-type. Le seul élément non immédiat dans cette preuve est qu'une fonction constante est continue.
En réduisant à homotopie près, nous obtenons notre élément neutre :

```
one := [const_x₀ X | X]
```

9.5 c_0 est l'élément neutre

On doit faire deux preuves très similaires, qui sont liées à la manière qu'à Lean de décrire un groupe :

- $[f] \cdot [c_0] = [f]$:
- $[c_0] \cdot [f] = [f]$

On ne traitera ici que la première preuve, la deuxième étant quasi-identique à la première. C'est une preuve souvent traitée avec peu de rigueur, souvent avec un schéma, et donc l'expression de l'homotopie à choisir est peu naturelle. On considère :

$$H(s,t) = \text{si } t \leq (1-s)/2 \text{ alors } x_0 \text{ , sinon } f((2-s)t - 1 + s)$$

Valeur aux extrémités : Étant donné que la composition est définie par partie, et que H l'est également, on fait appel 4 fois à la tactique `split_ifs`. C'est l'une des raisons expliquant la longueur de la preuve. L'utilisation combinée d'`exfalso` et de `linarith` permet de ne pas avoir à développer l'expression de la composition dans certains cas, et on a donc « seulement » 6 cas à traiter. Ici, contrairement à la preuve de la transitivité, on peut bien utiliser `linarith`, car les hypothèses sont souvent contradictoires sans appel à des résultats extérieurs.

Continuité de l'homotopie : Similairement à la preuve de la transitivité, on utilise `continuous_if`. La continuité sur les deux parties du carré $[0,1]^2$ est aisée, mais la démonstration de l'égalité des définitions à la frontière est plus difficile, et la preuve a été omise.

9.6 L'associativité de la composition

La preuve n'est pas très intéressante : elle est très longue car l'homotopie nécessaire enchaîne 2 définitions conditionnelles à la suite, mais elle n'utilise pas de techniques particulières. La preuve de la continuité n'a pas été traitée, mais outre les problèmes de frontières, elle ne poserait pas de problèmes.

9.7 $\bar{f}$ est inverse de f

Cette preuve a été omise par manque de temps. On procéderait de manière similaire aux autres, et outre les problèmes habituels de frontière et la longueur de la preuve liée à un découpage de l'intervalle en 3, elle ne serait pas particulièrement difficile.

10 La métaprogrammation et la tactique cont

Comme on peut facilement le remarquer, les preuves de continuité sont très répétitives. Cependant, un trait extrêmement puissant de Lean est sa capacité de métaprogrammation, i.e. qu'il est assez aisé d'améliorer Lean (en temps que langage) à partir de Lean. Ainsi, le dernier fichier, `tactics.lean`, contient une tactique (`cont`) qui permet d'automatiser en partie les preuves de continuité.

Cette tactique est une application brute-force des résultats de continuité qui sont fréquemment utilisés dans les fichiers. Si on se représente une preuve de continuité comme un arbre ayant pour feuilles la continuités des blocs de bases (constantes continues, fonction identité continue, la projection sur le premier élément est continue, etc.) et les nœuds internes sont les opérations continues sur plusieurs fonctions continues (la somme est continue, la multiplication est continue, etc.), le programme agit, dans une première étape, en parcourant tous les noeuds internes avec une profondeur p en paramètre, puis en concluant sur toutes les feuilles, dont le nombre est borné par le nombre d'objectifs encore ouverts.

Ainsi, cette tactique est efficace pour des fonctions simples, mais se trouve parfois être trop lente pour les résultats dont nous avons besoin. Il existe cependant un certain nombre d'optimisations possibles qui dépassent le cadre de ce TER. L'écriture de cette tactique a néanmoins eu des vertus pédagogiques et a permis de voir le concept de métaprogrammation dans Lean.

Troisième partie
Conclusion

Ce sujet permet de faire un état des lieux de la possibilité d'utiliser un assistant de preuve dans la pratique mathématique quotidienne. On en déduit qu'en dépit de la rigidité omniprésente de Lean et de sa difficulté à comprendre des résultats qui semblent triviaux, cet assistant possède (notamment grâce à mathlib) une bibliothèque vaste et facilement utilisable de résultats. Ainsi des constructions comme des espaces quotient et des démonstrations comme celles de continuité sont surprenamment très aisées à mettre en place.

Une telle remarque est particulièrement vraie pour la définition du groupe fondamental. C'est un objet relativement simple à comprendre, mais dont les détails de définitions sont longs à démontrer très rigoureusement. Il est donc remarquable qu'un tel objet puisse être défini dans un assistant de preuve, car c'est une définition « moins calcultoire » et « plus intuitive » que certains résultats d'algèbre pur, pour lequel l'ordinateur est particulièrement bien adapté.

La version prochaine de Lean, qui règle notamment les problèmes d'inférence de types, qui ralentissent et obfusquent les preuves, devrait cependant répondre à un certains nombres de critiques formulées précédemment.

Preuves omises

En dépit de la bonne définition du groupe fondamental dans Lean, on a omis un petit nombre de démonstration pour de multiples raisons :

Associativité de la composition : la preuve n'est pas complète, notamment par manque de temps.

Compatibilité de la composition avec le passage au quotient : Pour mener à bien cette démonstration, il faudrait considérer des homotopies à extrémités fixées, i.e. pour une homotopie H, on a $H(0,-) = x_0$ et $H(1,-) = x_0$. Cela n'a pas été le choix de conception initial, par soucis de généralité, mais cependant la preuve ne peut aboutir sans. Ainsi, pour démontrer entièrement ce résultat, il faudrait redéfinir l'homotopie à extrémités fixées. Ce n'est pas une tâche difficile, cependant elle implique de réécrire toutes les démonstrations.

Preuves de la frontière : Paradoxalement, certains résultats de topologie élémentaire sont difficiles à démontrer en l'état actuel de mathlib. C'est notamment le cas des démonstrations de frontières, car cela nécessite de déterminer l'intérieur et l'adhérence d'un ensemble qui est non seulement complexe à faire en toute rigueur mais également avec le manque de propriétés sur de telles construction dans mathlib.

Quelques `sorry` résiduels dans les preuves d'appartenance de sous-type : ces `sorry` sont souvent liés à une construction conditionnelle.
Lean possède la syntaxe `if p then x else y`, mais elle représente deux constructions différentes : `ite` et `dite`. Dans le premier cas, les définitions de x et de y ne dépendent pas de si p est vraie, contrairement au second cas. En toute rigueur, nous devrions utiliser `dite` dans toutes les définitions, cependant, la propriété `continuous_if` n'a pas encore d'équivalent pour `dite`. Or `ite` ne nous permet pas d'utiliser la véracité de p dans les définitions de x et y, et donc quand x et y sont des éléments de sous-types, on ne peux pas démontrer leur appartenance au sous-type.
En guise d'exemple, on peut donner les fonctions définies de la façon suivante :

```
ite (t.val≤0.5) (f(⟨2*t.val, sorry⟩) ) (g(⟨ 2*t.val-1, sorry⟩)
```

Les `sorry` sont nécessaire car on ne peut pas nommer l'hypothèse `t.val≤0.5` avec `ite`, et donc on ne peut pas l'utiliser pour montrer que `2*t.val` et `2*t.val-1` appartiennent à I.

Une raison pour les dernières preuves omises est que le système d'inférence de type dans Lean est pour le moment peu puissant, et que les choix d'utiliser massivement les typeclass dans mathlib ne s'accorde pas très bien avec l'état actuel du noyau de Lean. On pourra consulter [mC20] pour plus d'informations sur ce problème.

Pour aller plus loin

On peut finalement noter qu'il existe des méthodes pour approcher la topologie algébrique qui sont plus appropriées à une formalisation informatique. L'exemple de l'implémentation de HoTT (Théorie des types homotopiques) dans l'assistant Coq, qui

va bien plus loin que ce TER, tout en formalisant toutes les branches des mathématiques, montre que cela peut être une approche plus adaptée.

Références

[AdMK20] J. Avigad, L. de Moura, and S. Kong. *Theorem Proving in Lean.* 2020.

[Ite18] I. Itenberg. *Polycopié de Topologie algébrique.* 2018.

[Ler17] C. Leruste. *Topologie algébrique - Une introduction, et au delà.* 2017.

[mC20] The mathlib Community. The lean mathematical library. In *CPP2020*, 2020.

[Ned14] R. Nederpelt. *Type theory and formal proof : An introduction.* 2014.