Preuves assistées par ordinateur

Document 1 - Types et fonctions en LEAN

7 février 2025

I

Types et variables

Types des objets

En LEAN, comme dans beaucoup de langages, tous les objets ont un **type** bien défini. Cela permet à la machine de ne pas confondre un nombre entier avec une chaîne de caractères, par exemple.

Il est possible d'afficher le type d'un objet en tapant la commande `#check`.

Commandes	**Messages de la console**
`#check 2`	`2 : Nat`
`#check 3.4`	`3.4 : Float`
`#check "Bonjour tout le monde"`	`"Bonjour tout le monde" : String`

Comme on le voit au-dessus, quand on tape `#check`, suivi d'une expression, LEAN affiche l'expression, puis " :" , puis le **type** de l'expression.

Les types suivants sont classiques :

1. `Nat` : c'est le type des **entiers naturels**.
   Par exemple : `0`, `2`, `3242`.

2. `Int` : c'est le type des **entiers relatifs**.
   Par exemple : `-1`, `-1032`, `983`.

3. `Float` : c'est le type des **représentations décimales approchées d'un nombre réel** (c'est-à-dire des représentations décimales avec un nombre fini de chiffres après la virgule).
   Par exemple : `-30.4002`, `-932.0`, `234.01`.

4. `String` : c'est le type des **chaînes de caractères**, représentées par des expressions entre guillemets.
   Par exemple : `"Bonjour, je suis une chaîne de caractères"`.

Il existe d'autres types, que nous verrons au fur et à mesure des séances.

Variables

En LEAN, il est possible de déclarer des **variables**, pour demander à l'ordinateur de stocker en mémoire certaines expressions en leur donnant un nom.

LEAN aura besoin pour cela de connaître le **type** de la variable que l'on déclare.

Pour déclarer une variable, il faut utiliser le mot-clé `def`. La syntaxe est la suivante :

```
def nom_de_la_variable : type_variable := valeur_variable
```

Bien sûr, il faut remplacer les trois expressions `nom_de_la_variable`, `type_de_la_variable` et `valeur_de_la_variable` par celles qui sont pertinentes dans le contexte.

Par exemple, si on veut définir une variable « `numero_departement` » égale à 54, de type « entier naturel », on tape :

```
def numero_departement : Nat := 54.
```

Voici quelques exemples de déclarations de variables :

```
def nombre_planetes : Nat := 8

def record_temperature_au_pole_sud : Float := -82.8

def prenom_capitaine_haddock : String := "Archibald"
```

On peut imaginer que dans l'ordinateur, les variables sont stockées dans un grand tableau, qui indique à chaque fois le **nom** de la variable, son **type** et enfin sa **valeur**.

Nom de la variable	Type de la variable	Valeur de la variable
`nombre_planetes`	`Nat`	`8`
`record_temperature_au_pole_sud`	`Float`	`-82.8`
`prenom_capitaine_haddock`	`String`	`"Archibald"`

Quand des variables sont stockées en mémoire, il est possible de demander à LEAN les choses suivantes :

1. afficher le **type** d'une variable en tapant la commande `#check`, suivie du nom de la variable ;
2. afficher la **valeur** d'une variable en tapant la command `#print`, suivie du nom de la variable.

Par exemple :

Commandes	**Messages de la console**

`def nombre_planetes : Nat := 8`
`#check nombre_planetes`
`#print nombre_planetes`
	`nombre_planetes : Nat`
	`8`

`def bonjour_en_allemand : String :=`
`  "Guten Tag"`
`#check bonjour_en_allemand`
`#print bonjour_en_allemand`
	`bonjour_en_allemand : String`
	`"Guten Tag"`

On dispose aussi de la commande `#eval` qui peut servir à afficher le résultat de certains calculs, par exemple avec des nombres entiers.

Par exemple :

Commandes	**Messages de la console**
`#eval (3 * 8 + 2)`	26
`def nombre_planetes : Nat := 8`	
`#eval (nombre_planetes *  2)`	16

Détermination implicite du type

Parfois, LEAN est capable de deviner implicitement le type d'un objet. Par exemple :

Commandes	Messages de la console

`def a := 3`
 	
`#check a`
	`a : Nat`

Ici, on n'a pas indiqué le type de la variable a au moment de la déclarer, et LEAN a deviné automatiquement qu'il fallait donner le type `Nat` à la variable « a ».

On peut cependant vouloir donner à a un autre type. Par exemple, pour déclarer cette variable comme un entier relatif, on sait qu'il faut taper :

Commandes	Messages de la console

`def a : Int := 3`
 	
`#check a`
	`a : Int`

Résumé - Déclaration de variables

1. Pour déclarer une variable en LEAN, on utilise la syntaxe suivante :

   ```
   def nom_de_la_variable : type_variable := valeur_variable
   ```

   Par exemple :

```
def nombre_mystere : Nat := 5050.
```

2. Dans la mémoire, les variables sont stockées dans un tableau comme celui-ci :

Nom de la variable	Type de la variable	Valeur de la variable
nombre_mystere	Nat	5050
...	...	...

3. On peut afficher le **type** de la variable avec `#check` et sa **valeur** avec `#print`.

Commandes	Messages de la console
`#check nombre_mystere`	`nombre_mystere : Nat`
`#print nombre_mystere`	`5050`

II

Déclarations de fonctions

Fonctions

En LEAN, il est possible de déclarer des fonctions, qui prennent en entrée un certain nombre d'objets, et qui renvoient en sortie un objet. Le langage a besoin qu'on lui précise les **types** des objets en entrée et en sortie.

Il y a essentiellement deux syntaxes pour déclarer une fonction. Commençons par voir la première.

Déclaration de fonction - première syntaxe

Pour déclarer une fonction, il faut encore utiliser le mot clé `def`.

La première syntaxe que nous allons voir est la suivante :

Déclaration de fonction - version 1

```
def nom_de_la_fonction ( nom_entree : type_entree ) : type_sortie  :=
    expression_sortie
```

Là encore, il faut remplacer chacune des expressions `nom_de_la_fonction`, `nom_entree`, `type_entree`, `type_sortie` et `expression_sortie` par une expression adéquate.

Le retour à la ligne n'est pas nécessaire, mais peut-être utile pour bien organiser le code.

Attention à bien laisser chaque symbole « (», « : », «) » et « := » au même endroit que dans la syntaxe ci-dessus !

Voici deux exemples de déclarations de fonctions en utilisant la syntaxe précédente.

```
def carre (a : Nat) : Int :=
    a*a

def polynome (x : Int) : Rat :=
    3*x^2 + 10*x
```

Ici, la fonction `carre` représente la fonction suivante :

$$\begin{array}{ccc} \mathbb{N} & \to & \mathbb{Z} \\ a & \mapsto & a^2 \end{array}$$

La fonction `polynome` représente la fonction suivante :

$$\begin{array}{ccc} \mathbb{Z} & \to & \mathbb{Q} \\ x & \mapsto & 3x^2 + 10x \end{array}$$

Il est possible de donner en entrée un objet à une fonction, en écrivant le nom de l'objet à droite de la fonction. Si on veut afficher le résultat, on peut le faire avec la commande `#eval`.

Commandes	**Messages de la console**
`def ajoute_deux (x : Nat) : Nat :=` `  x+2`	
`#eval (ajoute_deux 2)`	4
`#eval (ajoute_deux 103)`	105

Il faut faire attention à fournir à la fonction un objet du type qu'elle attend. Si on lui donne un objet d'un autre type, on obtient une erreur.

Commandes	Messages de la console

`def ajoute_deux (x : Nat) : Int :=`
`  x+2`
`-- ajoute_deux prend en entrée un objet de type Nat`
 	
`#eval (ajoute_deux 3.4)`
	`Failed to synthesize`
	`OfScientific Nat`

Ici, le message apparaît puisqu'on a donné un nombre de type `Float` à la fonction `ajoute_deux`, alors qu'elle attendait un objet de type `Nat`.

Quel est le type d'une fonction ?

En LEAN, tous les objets ont un type. Cela signifie que si on déclare une fonction, elle doit aussi être d'un type particulier.

Pour voir à quoi ressemble ce type dans un cas particulier, utilisons `#check` pour l'afficher :

Commandes	**Messages de la console**

`def polynome (x : Nat) : Nat :=`
`  3*x^2 + 12`
 	
`#check (polynome)`
	
	`polynome : Nat→Nat`

Rappelez-vous : si a est une variable d'un certain type, alors « `#check` a » affiche le nom de la variable, suivi de « : », puis du type de la variable.

Cela signifie que `polynome` est de type `Nat→Nat`.

Quel est le type d'une fonction ? (Suite)

On sait que `#check` peut servir à afficher le **type** d'un objet, et que `#print` sert à afficher sa **valeur**. Que se passe-t-il si l'on fait `#print` sur une fonction ?

Commandes	**Messages de la console**

`def polynome (x : Nat) : Nat :=`
`  3*x^2 + 12`
 	
`#check (polynome)`
 	
`#print (polynome)`
	`polynome : Nat→Nat`
	
	`def polynome : Nat → Nat :=`
	`fun x => 3 * x ^ 2 + 12`

Ici, `#print` indique que `polynome` est un objet de type `Nat →Nat`, qui a la valeur « `fun x => 3 * x^2` ».

C'est la manière pour LEAN de dire que `polynome` représente la fonction qui va de $\mathbb{N}$ dans $\mathbb{N}$, qui à x associe $3x^2 + 12$.

Définition d'une fonction (deuxième syntaxe)

On dispose aussi d'une deuxième syntaxe pour définir une fonction. Voici cette nouvelle syntaxe :

Déclaration de fonction - version 2

```
def nom_de_la_fonction : type_entree → type_sortie  :=
                fun   nom_entree => expression_sortie
```

Rappelons que la première syntaxe était la suivante :

```
def nom_de_la_fonction ( nom_entree : type_entree ) : type_sortie  :=
    expression_sortie
```

Les deux manières de faire sont parfaitement synonymes l'une de l'autre. Dans la suite, on se servira surtout de la nouvelle syntaxe.

Donnons quelques exemples d'utilisation de la nouvelle syntaxe, en définissant quelques fonctions.

Commandes	**Messages de la console**
`def polynome_2 : Int → Int :=` `  fun x => (x + 1 )*(x + 23 )`	
`#eval (polynome_2  3 )`	52
`def autre_fonction : Int → Int :=` `  fun y => 2*(polynome_2 y)`	
`#eval (autre_fonction (-3) )`	-160

Avec ces exemples, la fonction `polynome_2` représente la fonction

$$\begin{array}{ccc} \mathbb{Z} & \to & \mathbb{Z} \\ x & \mapsto & (x+1)(x+23). \end{array}$$

La fonction `autre_fonction` représente la fonction

$$\begin{array}{ccc} \mathbb{Z} & \to & \mathbb{Z} \\ y & \mapsto & 2(y+1)(y+23). \end{array}$$

Exercice

Réécrivez les définitions de `polynome_2` et `autre_fonction`, en utilisant cette fois-ci la première syntaxe.

Nous allons faire dans un instant un petit récapitulatif sur ce qu'on a vu jusqu'ici. Avant cela, il nous reste une dernière chose à bien comprendre sur la deuxième syntaxe.

Complément sur la deuxième syntaxe - Stockage en mémoire d'une fonction

Rappelons que si l'on veut définir une *variable*, on peut utiliser la syntaxe suivante :

```
def nombre_mystere : Nat := 5050
```

Cela définit un objet dont le nom est « `nombre_mystere` », dont le type est « `Nat` », et dont la valeur est « 5050 ».

Comparons à la manière dont on peut définir une fonction :

```
def fonction_interessante : Nat → Int :=
    fun x => x^x + 1
```

En fait, il s'agit de la même syntaxe que pour définir une variable ! Le code ci-dessus définit un objet dont le nom est « `fonction_interessante`, dont le type est « `Nat→Int` », et dont la valeur est « `fun x => x^x + 1` ».

Utilisons des couleurs pour bien voir le parallèle.

```
def nombre_mystere : Nat := 5050
def fonction_interessante : Nat → Int :=
    fun x => x^x + 1
```

(En rouge : nom de l'objet, en vert : type de l'objet, en bleu : valeur de l'objet)

Dans la mémoire, les fonctions sont donc stockées dans le même tableau que les variables, mais leur type est un peu différent : il s'écrit avec une flèche « → ».

Nom de la variable	**Type de la variable**	**Valeur de la variable**
`nombre_mystere`	`Nat`	`5050`
`fonction_interessante`	`Nat →Int`	`fun x => x^x + 1`
...	...	...

Comme on le voit dans le tableau, la valeur aussi a une écriture particulière : elle utilise le mot-clé `fun`, qui indique que l'objet est une fonction.

Résumé - Déclaration de fonctions

1. Pour déclarer une fonction en LEAN, on a le choix entre deux syntaxes. On utilisera surtout la syntaxe suivante :

```
def nom_de_la_fonction : type_entree → type_sortie :=
                fun nom_entree => expression_sortie
```

   Par exemple :

```
def  fonction_carre  :  Float→Float  :=
                        fun x => x^2
```

2. Dans la mémoire, les fonctions sont stockées dans le même tableau que les autres variables, mais leur type indique qu'il s'agit de fonctions. La valeur a aussi une écriture particulière, utilisant le mot-clé `fun` :

Nom de la variable	Type de la variable	Valeur de la variable
`fonction_carre`	`Float→Float`	`fun x => x^2`
...	...	...

3. On peut appliquer une fonction à un objet du bon type, en écrivant le nom de cet objet après le nom de la fonction. On peut afficher le résultat avec `#eval`.

Commandes	Messages de la console
`#eval (fonction_carre 32.1)`	`1030.410000`

III

Fonctions de plusieurs variables

Comme dans le cas d'une fonction d'une seule variable, il y a deux syntaxes pour définir une fonction prenant en entrée plusieurs variables.

1. **Première possibilité** :

```
def nom_de_la_fonction ( nom_1 : type_1 ) ( nom_2 : type_2 ) ... (nom_n : type_n ) : type_sortie  :=
  expression_sortie
```

Bien sûr, il faut remplacer le nom de la fonction par celui de notre choix, et les noms et les types des entrées par ceux qui sont pertinents.

Par exemple, la fonction suivante prend un entier relatif x, puis un entier naturel y, et renvoie l'entier relatif donné par $x^y - x^3$.

```
def fonction_interessante (x : Int) (y : Nat) : Int :=
 x^y  - x^3
```

❷ Deuxième possibilité

```
def nom_de_la_fonction : type_1 → type_2 → ... → type_n → type_sortie :=
    fun  nom_1  nom_2  ...  nom_n => expression_sortie
```

Ainsi, on aurait pu définir la fonction `fonction_interessante` aussi de la façon suivante :

```
def fonction_interessante : Int → Nat →Int :=
    fun x y => x^y  - x^3
```

Prêtez attention au nombre de types à écrire dans la chaîne `Int→Nat→Int` : puisque la fonction prend en entrée 2 objets, il faut écrire $2 + 1 = 3$ types (autrement dit, les deux types d'entrées `Int` et `Nat`, et le type de sortie `Int`).

Application d'une fonction à plusieurs variables

Une fois définie une fonction de plusieurs variables, on peut l'appliquer à des objets. Pour cela, il faut écrire le nom de la fonction, puis les divers noms des objets auxquels on désire l'appliquer. On peut afficher le résultat avec `#eval`.

Commandes	**Messages de la console**
`def fonction_1 : Int → Nat → Int :=` `  fun x y => (x + 1 )*(y + 8 )`	
`#eval (fonction_1 (-3) 2)`	`-20`
`  -- ici on a (-3 + 1)*(2 + 8) = -20`	
`#eval (fonction_1 13  10)`	`252`
`  -- ici on a (13 + 1)*(10 + 8) = 252`	

Attention, il faut que les types des objets donnés en entrée correspondent à ceux des variables attendus par la fonction, sinon, on aura une erreur.

Commandes	**Messages de la console**

`def fonction_1 : Int → Nat → Int :=`
`  fun x y => (x + 1 )*(y + 8 )`
 	
`#eval (fonction_1 (-3) 2.1)`
`  -- Erreur : fonction_1 attend un objet Nat`
`  -- en deuxième variable`
	`Failed to synthesize`
	`  OfScientific Nat`

Avant de conclure ces notes, un mot sur le type des fonctions de plusieurs variables.
Comme précédemment, si on définit par exemple

```
def fonction_2 : Int → Nat →Int → Int :=
    fun x y z => x^y  - x^3 + z^2
```

alors « fonction_2 » est un objet de type « Int→Nat→Int→Int ».

Il est stocké en mémoire dans le même tableau que les autres variables, sous la forme suivante :

Nom de la variable	Type de la variable	Valeur de la variable
fonction_2	Int→Nat →Int→Int	fun x y z => x^y - x^3 + z^2
...	...	...

Résumé - Fonctions de plusieurs variables

1. Pour déclarer une fonction de plusieurs variables en LEAN, on a le choix entre deux syntaxes. On utilisera surtout la syntaxe suivante :

```
def nom_de_la_fonction : type_1 → type_2 → ... → type_n → type_sortie :=
                fun  nom_1  nom_2  ...  nom_n => expression_sortie
```

   Par exemple, la fonction suivante prend 3 entiers relatifs a, b, c et renvoie l'entier relatif $ab + c$:

```
def  fonction_du_TP1  :  Int→Int→Int→Int  :=
                    fun a b c => a*b + c
```

2. Dans la mémoire, les fonctions sont stockées dans le même tableau que les autres variables, mais leur type indique qu'il s'agit de fonctions. La valeur a aussi une écriture particulière, utilisant le mot-clé `fun` :

Nom de la variable	Type de la variable	Valeur de la variable
`fonction_du_TP1`	`Int→Int→Int→Int`	`fun a b c => a*b + c`
...	...	...

3. On peut appliquer une fonction à des objets du bon type, en écrivant les noms de ces objets après le nom de la fonction. On peut afficher le résultat avec `#eval`.

Commandes	Messages de la console
`#eval (fonction_du_TP1 (-2) 2 12 )`	8