UFR IM²AG
UNIVERSITÉ Grenoble Alpes

# Modèles de Calcul [ Lambda-Calcul ]
# Programmation en λ-calcul polymorphe

Pascal Fradet Jean-François Monin Catherine Parent-Vigouroux

*Dans la suite, on demandera de définir différentes fonctions. Il importera, dans le code que vous rendrez, d'ajouter sytématiquement des tests permettant de les vérifier en utilisant* `Compute`.

L'introduction du polymorphisme dans le système de types vu jusqu'ici en lambda-calcul permet de dépasser les limitations vues précédemment. Par exemple, on n'a plus besoin de fixer à l'avance un type destination pour les couples ; les opérations arithmétiques se définissent plus facilement. On peut même coder les types de données algébriques tels que les listes ou les arbres.

Pour former des types, on ajoute simplement la possibilité de *quantifier* (universellement) une variable de type. Par exemple : $\forall T, T \to T$.

## 1 Exemple simple : l'identité polymorphe

Intuitivement, la fonction identité $\lambda x.x$ s'applique à n'importe quoi (un booléen, un entier, un couple, une fonction sur les entiers, etc. ; autrement dit un habitant de *n'importe quel type*) et rend ce qui lui est donné en argument. Son type sera donc la généralisation de $T \to T$ à n'importe quel type T, c'est-à-dire $\forall T, T \to T$.

Cependant, avant d'appliquer la fonction identité `id` à un terme, on devra explicitement préciser le type de ce dernier. Par exemple, pour l'appliquer à l'entier naturel standard 3 de type `nat`, on écrira : `id nat` 3 au lieu de `id` 3. Autrement dit, la fonction `id` prend successivement deux arguments :

— un argument préliminaire pour indiquer un type
— l'argument principal ayant le type indiqué.

Pour distinguer ces deux sortes d'arguments, on garde traditionnellement λ pour les arguments principaux et on emploie Λ (un λ majuscule) pour les arguments de type. Dans le cas de l'identité, on obtient la définition suivante : $\texttt{id} \stackrel{\text{def}}{=} \Lambda T.\lambda x^T.x$ ou plus complètement : $\texttt{id}^{\forall T,T\to T} \stackrel{\text{def}}{=} \Lambda T.\lambda x^T.x$ ; l'application `id nat` donne donc $\lambda x^{\texttt{nat}}.x$ et ce terme peut alors être appliqué à 3.
La réduction complète est

$$(\Lambda T.\lambda x^T.x)\,\texttt{nat}\,3 \quad\longrightarrow\quad (\lambda x^{\texttt{nat}}.x)\,3 \quad\longrightarrow\quad 3$$

En Coq, on garde la même notation `fun ... =>` pour l'abstraction sur les types Λ comme pour l'abstraction sur les termes λ. Pour la quantification sur une variable de type dans la définition d'un type, la notation ∀ s'écrit en Coq `forall`. Ainsi, l'identité polymorphe se définit ainsi (**attention à la virgule après** `forall T: Set`) :

```
(* Type de l'identite polymorphe *)
Definition tid : Set := forall T: Set, T -> T.
Definition id : tid := fun T:Set => fun x:T => x.
```

**ATTENTION**
**1) il faut appeler Coq avec une option particulière pour permettre l'usage du typage polymorphe :**
**$ coqide -impredicative-set**
**2) Bien indiquer le type « Set » pour `tid`, pour `T` et pour tous les types à venir.**

1. Tester en Coq l'identité polymorphe sur des entiers standard (`nat`) et des booléens standard (`bool`).
2. Définir une fonction de `bool` vers `nat`, puis lui appliquer l'identité polymorphe.
   Par exemple, on peut définir une fonction des booléens vers les entiers, associant l'entier 1 à `true` et l'entier 0 à `false` – c'est le nombre de `true` parmi *un* booléen! Une telle fonction est définie de la façon suivante en Coq :

```
Definition nbtrue1 := fun b =>
  match b with
  | true => 1
  | false => 0
  end.
```

3. Vérifier que l'on peut appliquer la fonction `id` à... elle-même!
   (Il faut préalablement lui appliquer un argument intermédiaire bien choisi ; mais cet argument indique un type et non un λ-terme, donc il n'intervient pas dans le « véritable » calcul ; dit autrement, le λ-terme non typé sous-jacent est bien $(\lambda x.x)(\lambda x.x)$).
4. Démontrer un théorème indiquant que la fonction `id` rend ce qui lui est donné en argument.

```
Theorem th_id : forall T: Set, forall x: T, id T x = x.
```

## 2 Booléens avec typage polymorphe

On s'intéresse ici au codage purement fonctionnel (en lambda-calcul) des booléens, et non aux booléens standards prédéfinis en Coq (type `bool`).

Le type polymorphe des booléens en lambda-calcul est $\texttt{pbool} \stackrel{\text{def}}{=} \forall \mathrm{T}.\, \mathrm{T} \to \mathrm{T} \to \mathrm{T}$, ses constructeurs sont $\texttt{ptr} \stackrel{\text{def}}{=} \Lambda \mathrm{T}.\, \lambda x^{\mathrm{T}} y^{\mathrm{T}}.\, x$ et $\texttt{pfa} \stackrel{\text{def}}{=} \Lambda \mathrm{T}.\, \lambda x^{\mathrm{T}} y^{\mathrm{T}}.\, y$.

1. Définir en Coq `pbool`, `ptr` et `pfa`.
2. Coder en Coq la négation d'un booléen `pbool` selon deux méthodes :
   — pour la première, on prend le même λ-terme pur sous-jacent qu'auparavant (en typage simple), qui est $\lambda b.\lambda x\, y.b\ y\ x$ – et on complète par les indications de typage appropriées ;
   — pour la seconde, on prend un autre λ-terme pur sous-jacent, représentant intuitivement « **if** $b$ **then** false **else** true », qui est $\lambda b.b(\lambda x\, y.y)(\lambda x\, y.x)$ – et on complète là aussi par les indications de typage appropriées, `pbool` étant lui-même de type `Set`.
3. Coder en Coq la conjonction et la disjonction des booléens `pbool`.
4. Définir en Coq une fonction qui prend en argument un booléen $b$ de type `pbool` et qui rend l'entier 3 ou l'entier 5 (de type `nat`) suivant que $b$ est vrai ou faux.

## 3 Produits

La notion de produit de types correspond à l'idée de juxtaposition de types de données.
Le type polymorphe des couples d'éléments respectivement de type U et V est
$\mathrm{U} \times \mathrm{V} \stackrel{\text{def}}{=} \forall \mathrm{T}, (\mathrm{U} \to \mathrm{V} \to \mathrm{T}) \to \mathrm{T}$.
Un couple $(u, v)$ sera alors codé par $\Lambda \mathrm{T}.\, \lambda k^{\mathrm{U} \to \mathrm{V} \to \mathrm{T}}.\, k\ u\ v$.
Pour utiliser un couple codé ainsi, on lui passe deux arguments : le premier est le type du résultat attendu, le second est une continuation qui est une fonction prenant en arguments les deux éléments du couple, et qui construit le résultat attendu à partir de ces derniers.
Vérifier que $(\Lambda \mathrm{T}.\, \lambda k^{\texttt{nat} \to \texttt{nat} \to \mathrm{T}}.\, k\ 3\ 5)\ \texttt{nat}\ (\lambda x\, y.\, x + y)$ se réduit bien en 8.

1. Coder en Coq le type `pprod_nb` des couples constitués d'un `nat` et d'un `bool` ainsi que le constructeur de couples correspondant `pcpl_nb`.
2. Coder en Coq le type `pprod_bn` des couples constitués d'un `bool` et d'un `nat` ainsi que le constructeur de couples correspondant `pcpl_bn`.

3. Coder en Coq une fonction de `pprod_nb` vers `pprod_bn` qui échange les deux éléments du couple en argument. Tester cette fonction.
   *Penser à réutiliser les fonctions définies précedemment.*
4. Il est plus pratique de coder une fois pour toutes en Coq le type des couples constitués d'un habitant de U et d'un habitant de V, où U et V sont deux types. On aura donc un type paramétré par deux types U et V. Une telle définition s'écrit en commençant ainsi :
   `Definition pprod : Set -> Set -> Set := fun U V => forall T:Set, etc.`
   Ou bien, de façon équivalente :
   `Definition pprod (U V: Set) : Set := forall T:Set, etc.`
   Donner la définition complète de `pprod` en Coq ainsi que celle du constructeur de couples correspondant `pcpl`.

# 4 Sommes (optionnel, non indispensable pour les entiers)

La notion de somme de types correspond à l'idée de **choix** entre types de données.

## 4.1 Entier optionnel

Un entier optionnel, c'est le choix entre soit une donnée de type entier, soit l'absence de donnée. Son type polymorphe est `opnat` $\stackrel{\text{def}}{=} \forall \mathrm{T}, (nat \to \mathrm{T}) \to \mathrm{T} \to \mathrm{T}$. Les constructeurs associés sont

— `Some_nat` de type `nat` → `opnat`, défini par $\lambda n.\, \Lambda \mathrm{T}.\, \lambda k_1^{\texttt{nat}\to\mathrm{T}}.\, \lambda k_2^{\mathrm{T}}.\, k_1\ n$

— `No_nat` de type `opnat`, défini par $\Lambda \mathrm{T}.\, \lambda k_1^{\texttt{nat}\to\mathrm{T}}.\, \lambda k_2^{\mathrm{T}}.\, k_2$

Vérifier que (`Some_nat`3)`nat`$(\lambda n^{\texttt{nat}}.\ n+n)\,1$ se réduit bien en 6, et que `No_nat``nat`$(\lambda n^{\texttt{nat}}.\ n+n)\,1$ se réduit bien en 1.

1. Donner en Coq les définitions du type `opnat` et de ses constructeurs.
2. Donner en Coq une fonction qui prend un argument $s$ de type `opnat` qui rend un entier valant $2n$ si $s$ est `Some_nat` $n$, et valant 1 si $s$ est `No_nat`.
3. Donner en Coq une fonction qui prend un argument $s$ de type `opnat` qui rend un `opnat` valant `Some_nat` (S $p$) si $s$ est `Some_nat` $p$, et valant `Some_nat` 0 si $s$ est `No_nat`.

## 4.2 Choix entre (ou somme de) deux types de données

Le type polymorphe des sommes d'éléments de type U ou de type V est
$\mathrm{U}+\mathrm{V} \stackrel{\text{def}}{=} \forall \mathrm{T}, (\mathrm{U} \to \mathrm{T}) \to (\mathrm{V} \to \mathrm{T}) \to \mathrm{T}$.
Le constructeur de U + V à partir d'un élément $u$ de U est codé par $\Lambda \mathrm{T}.\, \lambda k_1^{\mathrm{U}\to\mathrm{T}}.\, \lambda k_2^{\mathrm{V}\to\mathrm{T}}.\, k_1\ u$.
Le constructeur de U + V à partir d'un élément $v$ de V est codé par $\Lambda \mathrm{T}.\, \lambda k_1^{\mathrm{U}\to\mathrm{T}}.\, \lambda k_2^{\mathrm{V}\to\mathrm{T}}.\, k_2\ v$.

1. Donner en Coq les définitions correspondantes :
   ```
   Definition psom (U V: Set) : Set := forall T:Set, etc.
   Definition inj1 (U V: Set) : U -> psom U V := fun u => fun T:Set => etc.
   Definition inj2 (U V: Set) : V -> psom U V := etc.
   ```
2. Définir en Coq une fonction qui prend en argument un élément $x$ de type `nat`+`bool` (choix entre `nat` et `bool`) et qui rend :
   — le double de $n$ si $x$ provient de l'entier naturel $n$ ;
   — 1 si $x$ provient du booléen `true` ;
   — 0 si $x$ provient du booléen `false`.
   On peut utiliser la fonction `nbtrue1` vue précédemment.

# 5 Entiers de Church avec typage polymorphe

Le type polymorphe des entiers de Church est $\texttt{pnat} \stackrel{\text{def}}{=} \forall \mathrm{T}, (\mathrm{T} \to \mathrm{T}) \to (\mathrm{T} \to \mathrm{T})$.
Ses constructeurs sont :
$\texttt{p0} : \texttt{pnat} \stackrel{\text{def}}{=} \Lambda \mathrm{T}. \lambda f^{\mathrm{T}\to\mathrm{T}} x^{\mathrm{T}}. x$
$\texttt{pS} : \texttt{pnat} \to \texttt{pnat} \stackrel{\text{def}}{=} \lambda n^{\texttt{pnat}}. \Lambda \mathrm{T}. \lambda f x. f\ (n\ \mathrm{T}\ f\ x)$

## 5.1 Opérations simples

1. Définir les opérations d'addition, de multiplication et de test à 0 en adaptant les termes obtenus au cours des sessions antérieures.
2. Une nouvelle version possible de l'addition est
   $\texttt{pplus} : \texttt{pnat} \to \texttt{pnat} \to \texttt{pnat} \stackrel{\text{def}}{=} \lambda n^{\texttt{pnat}} m^{\texttt{pnat}}. n\ \texttt{pnat}\ \texttt{pS}\ m$.
   Donner une interprétation intuitive de cette définition puis la coder en Coq et la tester sur quelques exemples.

## 5.2 Prédécesseur (optionnel)

Le calcul du précédesseur d'un entier $n$ nécessite l'utilisation de structures de données vues dans les sections 3 ou 4. Les deux méthodes vues en λ-claculnon typé sotn utilsiables. La première utilise la notion de produit, la seconde celle de somme.

1. Méthode 1 : l'idée est d'itérer $n$ fois une fonction agissant sur des couples d'entiers – à partir de $(x, y)$ donné en argument, cette fonction rend $(y, \mathrm{S}y)$ ; en itérant $n$ fois cette fonction sur $(0, 0)$, on obtient $(n-1, n)$ et il suffit d'extraire la première composante de ce couple.
2. Méthode 2 : l'idée est d'itérer $n$ fois une fonction agissant sur des entiers polymorphes optionnels, à définir sur le modèle de la section 4.1, plus exactement la fonction décrite en question 3 de cette section, mais en remplaçant `nat` par `pnat` ; à la fin, extraire du résultat obtenu l'entier s'il y en a un, ou retourner 0 s'il n'y en a pas.