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, soit en utilisant `Compute`*, soit en utilisant des théorèmes simples énonçant que telle fonction appliquée à tel(s) argument(s) rend tel résultat attendu. Cette fiche est longue, il est recommandé d'en faire le plus possible pour atteindre des programmes sur les entiers tels que le prédécesseur ou la factorielle. A cet effet, seule la section 4.1 de la partie 4 est indispensable. Il n'est pas indispensable non plus d'avoir fait la totalité de la partie 3 – mais il est bon d'avoir atteint au moins la question 3 de cette partie.*

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 $\forall$ 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.
   ```

Remarque : imprédicativité

Un type tel que `tid` est de la forme $\forall T, etc.$ ou plus précisément $\forall T^{\texttt{Set}}, etc.$ On remarque que `tid` est lui même de type `Set`, autrement dit, la définition de `tid` comporte une quantification sur des types dont `tid` fait lui-même partie. On dit alors que le système de types considéré est *imprédicatif*. C'est l'imprédicativité qui, au niveau des termes, donne la possibilité d'appliquer la fonction `id` à elle-même.

D'un point de vue calculatoire, l'idée d'un programme pouvant se prendre lui-même en argument n'a rien de choquant. Pensons à un compilateur C écrit en C. Ou, dans un autre modèle de calcul, à une machine de Turing universelle. En revanche, on ne peut pas représenter ces fonctions par des fonctions du cadre habituel de la théorie des ensembles (sans entrer dans les détails, il existe des constructions mathématiques plus compliquées qui conviennent).

Nous verrons plus loin d'autres exemples de fonctions (λ-termes) pouvant s'appliquer à elles-mêmes, utilisant là encore l'imprédicativité du système de types.

2 Booléens avec typage polymorphe

Pour cette section, il vous est demandé de vous munir de l'énoncé (et de vos solutions) de la fiche consacrée aux booléens avec typage simple (en fait trop simple) : plusieurs λ-termes purs sous-jacents sont les mêmes ; lorsque c'est le cas, il est important d'observer les différences dans le typage, et sinon, c'est parce que le typage polymorphe offre la possibilité de donner un type à des termes qui n'en avaient pas auparavant – on a donc gagné en pouvoir d'expression et il est important de bien le voir.

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 `pbool` $\stackrel{\text{def}}{=} \forall T.\, T \to T \to T$, ses constructeurs sont `ptr` $\stackrel{\text{def}}{=} \Lambda T.\, \lambda x^T y^T.\, x$ et `pfa` $\stackrel{\text{def}}{=} \Lambda T.\, \lambda x^T y^T.\, y$.

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 $\lambda b.\lambda xy.b\ y\ x$ – 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 $\lambda b.b(\lambda xy.y)(\lambda xy.x)$ – et on complète là aussi par les indications de typage appropriées, requérant ici une utilisation de l'imprédicativité du système, `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.
5. (**) En utilisant l'imprédicativité, définir en Coq une fonction qui prend en argument un booléen b de type `pbool` et qui rend b appliqué à lui-même.
   Donner une interprétation intuitive simple à la fonction obtenue, utilisant des connecteurs booléens.

3 Produits

La notion de produit de types correspond à l'idée de juxtaposition de types de données. L'ordre est important : un entier suivi d'un booléen, ce n'est pas la même chose qu'un booléen suivi d'un entier.
Le type polymorphe des couples d'éléments respectivement de type U et V est
$U \times V \stackrel{\text{def}}{=} \forall T, (U \to V \to T) \to T$.
Un couple (u, v) sera alors codé par $\Lambda T.\, \lambda k^{U\to V\to T}.\, k\ u\ v$.
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.
Par exemple, considérons le couple d'entiers $(3, 5)$, codé par $c \stackrel{\text{def}}{=} \Lambda T.\, \lambda k^{\texttt{nat}\to\texttt{nat}\to T}.\, k\ 3\ 5$. Pour calculer la somme de ses deux éléments, on lui passe en argument `nat` (le type du résultat attendu), puis la continuation $\lambda x\, y.\, x + y$, ce qui donne : $c\ \texttt{nat}\ (\lambda x\, y.\, x + y)$.
Vérifier que $(\Lambda T.\, \lambda k^{\texttt{nat}\to\texttt{nat}\to T}.\, k\ 3\ 5)\ \texttt{nat}\ (\lambda x\, y.\, x + y)$ 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 `ppair_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 `ppair_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 `ppair`.
5. Observer que les exemples précédents de couples constitués d'un `nat` et d'un `bool`, ou inversement peuvent être respectivement typés et codés en utilisant `pprod` et `ppair`.
6. Donner en Coq des exemples de couples constitués d'un `nat` et d'un `bool`. Observer leurs valeurs en utilisant `Compute`.

7. Coder en Coq une fonction générale qui échange les 2 éléments d'un couple donné en argument.
8. (*) Prouver en Coq qu'en appliquant successivement 2 fois la fonction précédente, on retrouve le couple initial.

4 Sommes

La notion de somme de types correspond à l'idée de choix entre types de données. On commence par deux exemples, puis on introduit la somme de deux types généraux U et V.

4.1 Entier optionnel

Un entier optionnel, c'est le choix entre soit l'absence de donnée, soit une donnée de type entier. Son type polymorphe est `opnat` $\stackrel{\text{def}}{=} \forall T, T \to (nat \to T) \to T$. Les constructeurs associés sont
— `No_nat` de type `opnat`, défini par $\Lambda T.\, \lambda k_1^T.\, \lambda k_2^{\texttt{nat}\to T}.\, k_1$
— `Some_nat` de type `nat` → `opnat`, défini par $\lambda n.\, \Lambda T.\, \lambda k_1^T.\, \lambda k_2^{\texttt{nat}\to T}.\, k_2\ n$
Pour utiliser un terme s de type `opnat` on lui donne donc un argument pour le type de sortie, puis deux continuations : la première est la valeur rendue dans le cas où s est `No_nat`, la seconde est la valeur rendue dans le cas où s est `Some_nat` n – cette continuation est une fonction de n. Par exemple, si on veut rendre un entier valant 1 si s est `No_nat`, ou qui vaut $2n$ si s est `Some_nat` n, on écrit :
$s\ \texttt{nat}\ 1\ (\lambda n^{\texttt{nat}}.\ n+n)$.
Vérifier que `No_nat` `nat` $1\ (\lambda n^{\texttt{nat}}.\ n+n)$ se réduit bien en 1, et que $(\texttt{Some\_nat}\ 3)\ \texttt{nat}\ 1\ (\lambda n^{\texttt{nat}}.\ n+n)$ se réduit bien en 6.

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

4.2 Choix entre un, deux ou trois booléens

1. On donne l'exemple d'un type somme de un ou deux ou trois booléens.

```
(* Somme a trois choix entre 1 booleen, 2 booleens, 3 booleens *)
Definition b123 :=
  ∀ T: Set,
  (bool → T) →
  (bool → bool → T) →
  (bool → bool → bool → T) → T.
```

   Dans le cas de notre exemple, on doit définir les 3 constructeurs correspondants aux lambda-termes non typés suivants :
   — $\lambda a.\lambda k_1 k_2 k_3.k_1\ a$,
   — $\lambda a.\lambda b.\lambda k_1 k_2 k_3.k_2\ a\ b$,
   — $\lambda a.\lambda b.\lambda c.\lambda k_1 k_2 k_3.k_3\ a\ b\ c$.
   Définir en Coq ce type `b123`, ainsi que les constructeurs correspondants.
2. Toute définition de fonction sur un tel type somme prévoit autant de continuations que de cas dans le type somme. On veut définir la fonction qui calcule combien d'éléments ont la valeur `true` dans un type somme de un, deux ou trois booléens. On réutilisera la fonction `nbtrue1` vue plus haut. Le lambda-terme correspondant à la fonction utilise alors 3 continuations différentes pour les trois cas :

$\lambda t.(t\ (\lambda a.(\texttt{nbtrue1}\ a))$
$\quad(\lambda a.\lambda b.(\texttt{nbtrue1}\ a) + (\texttt{nbtrue1}\ b))$
$\quad(\lambda a.\lambda b.\lambda c.(\texttt{nbtrue1}\ a) + (\texttt{nbtrue1}\ b) + (\texttt{nbtrue1}\ c)))$

Coder en Coq ce lambda-terme, puis faire un calcul au moins sur un exemple pour valider la définition de la fonction ci-dessus.

4.3 Choix entre deux types de données

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

1. Donner en Coq les définitions correspondantes :
   ```
   Definition psom (U V: Set) : Set := forall T:Set, etc.
   Definition C1 (U V: Set) : U -> psom U V := fun u => fun T:Set => etc.
   Definition C2 (U V: Set) : V -> psom U V := etc.
   ```
2. Définir en Coq une fonction qui prend en argument un élément x de type couple `(nat,bool)` et 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.
3. (**) Définir en Coq une fonction qui prend en argument un élément x de type couple `(nat,bool)` et rend :
   — la représentation de (true, n) si x provient de l'entier naturel n ;
   — la représentation de (false, 1) si x provient du booléen `true` ;
   — la représentation de (false, 0) si x provient du booléen `false`.

5 Entiers de Church avec typage polymorphe

Pour cette section, il vous est demandé de vous munir de l'énoncé (et de vos solutions) de la fiche consacrée aux entiers de Church avec typage simple (en fait trop simple) : plusieurs λ-termes purs sous-jacents sont les mêmes ; lorsque c'est le cas il est important d'observer les différences dans le typage, et sinon, c'est parce que le typage polymorphe offre la possibilité de donner un type à des termes qui n'en avaient pas auparavant – on a donc gagné en pouvoir d'expression et il est important de bien le voir.

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

1. Définir les opérations d'addition, de multiplication et de test à 0 en adaptant les termes donnés dans la fiche précédente sur les entiers.
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.
3. Calcul du prédécesseur d'un entier n : on propose ici deux méthodes, la première utilise la notion de produit, la seconde celle de somme.
   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, Sy) ; 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.
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.
Coder en Coq le calcul du prédécesseur suivant l'une au moins de ces méthodes.

4. Soustraction : pour calculer $n - m$ une idée possible est d'itérer m fois la fonction prédecesseur sur n.
   Coder en Coq le calcul de la soustraction.
5. Comparaison : pour déterminer si $n \leq m$, une idée possible est de calculer $n - m$ et d'examiner si le résultat est nul ;
   Coder en Coq le calcul de $n \leq m$.
6. Fonction factorielle : coder en Coq le calcul de $n! = 1 \times 2 \times \ldots n$. On pourra s'inspirer de l'astuce utilisée pour le calcul du prédécesseur (itérer une fonction bien choisie entre des couples d'entiers).
7. (**) En utilisant l'imprédicativité, définir en Coq une fonction qui prend en argument un entier n de type `pnat` et qui rend n appliqué à lui-même.
   Effectuer quelques évaluations pour $n = 0, 1, 2...$ et donner une interprétation intuitive simple à la fonction obtenue.