UFR IM²AG
UNIVERSITÉ Grenoble Alpes

Modèles de Calcul [Lambda-Calcul]
Projet à rendre avant le 07/04

Martin Bodin, Pascal Fradet, Jean-François Monin

Ce projet, à réaliser par équipes de 2 ou 3, comprend deux parties avec de nombreux bonus. Les questions en bonus, destinées à illustrer des techniques de programmation en λ-calcul plus avancées, peuvent généralement être traitées indépendamment les unes des autres et pourront donner si besoin des points supplémentaires à la note globale de la matière.

Noter que la taille du code à écrire est en fait très courte (beaucoup plus que l'énoncé!), mais attention, la clarté du code et la qualité des explications ou observations en commentaires seront prises en compte.

La première partie reprend essentiellement les exercices de pratique Coq vus dans les fiches 2 (λ-calcul non typé) et 3 (λ-calcul simplement typé). Pour obtenir une note correcte, il suffira de traiter parfaitement cette partie.

La seconde partie vise un codage des listes (et des arbres binaires en super bonus).

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.*

Le barème est indicatif.

1 Partie 1 (12 pts)

1.1 λ-calcul non typé

Rendre un premier fichier Coq compilable (utilisant explicitement la bibliothèque `untypedLC`) qui suit l'ordre indiqué.

1. Booléens (codage des constantes et des opérations de base)
2. Entiers naturels (codage de quelques constantes, des opérations successeur, addition et multiplication, et du test à 0)
3. Couples
4. *Structure de choix (inj1, inj2, donnée optionnelle) : bonus*
5. Prédécesseur (cpred ; *cpredo en bonus*)
6. Factorielle

1.2 λ-calcul simplement typé

Rendre un second fichier Coq compilable.

1. Booléens (codage des constantes et des opérations de base)
2. Entiers naturels (codage de quelques constantes, des opérations successeur, addition et multiplication, et du test à 0)

2 Partie 2 : programmation de structures avancées en λ-calcul (8 pts + bonus)

L'objectif de cette partie est de programmer sur des listes (voire des arbres binaires) au moyen d'un codage en λ-calcul pur. Il est possible de le faire en λ-calcul non typé mais le typage constitue une aide précieuse. Cependant le λ-calcul simplement typé est trop pauvre, on a besoin du *polymorphisme* introduit ici de manière progressive.

Motivation intuitive

Le type simple pour les booléens donné auparavant $T \to T \to T$ a la bonne structure, mais on aimerait pouvoir y remplacer à volonté T par un type quelconque. En effet, étant donné une expression booléenne B, il est souhaitable de pouvoir l'intégrer aussi bien dans une expression entière comme **if** B **then** 2 **else** 3 (ici T est remplacé par `nat`) ou que dans une expression booléenne comme **if** B **then** false **else** true (ici T est remplacé par `bool`), et on peut envisager bien d'autres utilisations où T serait remplacé par d'autres types encore.

Et c'est exactement ce qu'autorise le λ-calcul typé polymorphe. Ainsi le type des booléens polymorphes s'écrit : $\forall T, T \to T \to T$, ce qui indique qu'un booléen polymorphe attend un premier argument supplémentaire qui annonce quel est le type T à utiliser (qui serait `nat` ou `bool` dans les exemples précédents). Cela va être détaillé dans la suite, en commençant par un exemple encore plus simple que les booléens.

2.1 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 $\forall T, T \to T$.

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

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`). Exemple : `Compute id nat 3.`

2. Définir une fonction de `bool` vers `nat` et 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 là encore lui appliquer préalablement un argument intermédiaire bien choisi (`tid`) qui annonce le type du terme à suivre (`id`), ce qui donne : `id tid id`.

2.2 Booléens avec typage polymorphe

On s'intéresse ici 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` $\stackrel{\text{def}}{=} \forall \mathrm{T}, \mathrm{T} \to \mathrm{T} \to \mathrm{T}$, ses constructeurs sont `ptr` $\stackrel{\text{def}}{=} \Lambda \mathrm{T}.\lambda x^{\mathrm{T}} y^{\mathrm{T}}.\, x$ et `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 que dans la fiche 3, qui est $\lambda b.\lambda x y.b\ y\ x$ et on complète par les indications de typage appropriées : $\lambda b.\Lambda \mathrm{T}.\lambda x^{\mathrm{T}} y^{\mathrm{T}}.\, b\ \mathrm{T}\ y\ x$;
   — 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.
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. (*bonus*) 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.

2.3 Structures de données : couples et choix

2.3.1 Couples (produits de types)

Le type polymorphe des couples d'éléments respectivement de type A et de type B est
$\mathrm{A} \times \mathrm{B} \stackrel{\text{def}}{=} \forall \mathrm{T}, (\mathrm{A} \to \mathrm{B} \to \mathrm{T}) \to \mathrm{T}$.
Un couple (a, b) sera alors codé par $\Lambda \mathrm{T}.\lambda k^{\mathrm{A}\to\mathrm{B}\to\mathrm{T}}.\, k\ a\ b$.

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.
4. Il est plus pratique de coder une fois pour toutes en Coq le type des couples constitués d'un habitant de A et d'un habitant de B, où A et B sont deux types. On aura donc un type paramétré par deux types A et B. Une telle définition s'écrit en commençant ainsi :
   `Definition pprod : Set -> Set -> Set := fun A B => forall T:Set, etc.`
   Ou bien, de façon équivalente :
   `Definition pprod (A B: Set) : Set := forall T:Set, etc.`
   Donner la définition complète de `pprod` en Coq ainsi que celle du constructeur de couples correspondant `pcpl`.

2.3.2 Choix (sommes de types)

Le type polymorphe des termes qui sont soit de type A soit de type B est
$A + B \stackrel{\text{def}}{=} \forall T, (A \to T) \to (B \to T) \to T$.
Le constructeur de $A + B$ à partir d'un élément a de A est codé par $\Lambda T.\, \lambda k_1^{A\to T}.\, \lambda k_2^{B\to T}.\, k_1\ a$.
Le constructeur de $A + B$ à partir d'un élément v de V est codé par $\Lambda T.\, \lambda k_1^{A\to T}.\, \lambda k_2^{B\to T}.\, k_2\ b$.

Donner en Coq les définitions correspondantes :

```
Definition psom (A B: Set) : Set := forall T:Set, etc.
Definition inj1 (A B: Set) : A -> psom A B := fun u => fun T:Set => etc.
Definition inj2 (A B: Set) : B -> psom A B := etc.
```

Définir en Coq une fonction `toutvr` de type `psom pbool` (`pprod pbool pbool`) → `pbool` qui rend `cfa` si et seulement si tous les booléens contenus dans la valeur en entrée sont `cfa`.

2.4 Entiers de Church avec typage polymorphe

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.
2. *Bonus* 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 : l'idée est d'itérer n fois une fonction agissant sur des couples d'entiers.
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. *Bonus* 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. *Bonus* 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. *Bonus* 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.

2.5 Listes (bonus)

Au lieu de $\forall T, (T \to T) \to T \to T$, une autre représentation possible des entiers utiliserait le type
$\forall T, T \to (T \to T) \to T$.

Ce dernier type peut se lire sous forme d'une somme entre
— la construction `p0` d'un entier de base (intuitivement : l'entier 0)
— la construction `pS` d'un nouvel entier à partir d'un entier existant (intuitivement : l'entier « suivant », ou son successeur).

Un tel entier attend, après un type destinataire (T), deux continuations :
— la première, appelée x, sans argument, donnant une valeur attendue pour le cas de l'entier 0 ;
— la seconde, appelée f, pour le cas du successeur d'un entier n, à appliquer *non pas* à n, mais au *résultat* de la même fonction appliquée récursivement à n.

Ainsi, l'entier p0, appliqué à ces deux continuations, produira x ; l'entier pS p0, appliqué à ces deux continuations, produira $f\ x$; l'entier pS (pS p0), appliqué à ces deux continuations, produira $f\ (f\ x)$; et ainsi de suite.

La définition utilisée auparavant ($\forall$T, (T → T) → T → T), peut s'expliquer de la même manière mais avec les deux constructions présentées dans l'ordre inverse, ce qui a dans ce cas précis quelques avantages techniques, par exemple une définition très concise de la multiplication.

Pour les listes, on suit une approche analogue. Le type polymorphe des listes d'entiers est :
$\texttt{listen} \stackrel{\text{def}}{=} \forall \text{T}, \text{T} \to (\texttt{pnat} \to \text{T} \to \text{T}) \to \text{T}$.
Plus généralement, le type polymorphe des listes d'éléments de A, où A est un type, est
$\texttt{liste}\,\text{A} \stackrel{\text{def}}{=} \forall \text{T}, \text{T} \to (\text{A} \to \text{T} \to \text{T}) \to \text{T}$.
On peut lire ce type comme une somme entre

— rien (la liste vide)

— un couple constitué d'un habitant de A et d'une liste.

Ses constructeurs sont :

— $\texttt{pnil}\,\text{A} : \texttt{liste}\,\text{A} \stackrel{\text{def}}{=} \Lambda\text{T}.\lambda x^{\text{T}} c^{\text{A}\to\text{T}\to\text{T}}.\ x$

— $\texttt{pcons}\,\text{A} : \text{A} \to \texttt{liste}\,\text{A} \to \texttt{liste}\,\text{A} \stackrel{\text{def}}{=} \lambda a^{\text{A}} q^{\texttt{liste}\,\text{A}}.\Lambda\text{T}.\lambda x^{\text{T}} c^{\text{A}\to\text{T}\to\text{T}}.\ c\ a\ (q\ \text{T}\ x\ c)$.

Une liste est donc une fonction qui attend, après un type destinataire (T), deux continuations :

— la première, appelée x, sans argument, donnant une valeur attendue pour le cas de la liste vide ;

— la seconde, appelée c, pour le cas d'une liste constituée d'une tête a et d'une queue q, à appliquer à deux arguments fournis successivement, à savoir a et *non pas* la queue q, mais le *résultat* de la même fonction appliquée récursivement à q.

Ainsi la liste pcons A a_1 (pcons A a_2 (pcons A a_3 (pnil A))), appliquée à ces deux continuations, produira le calcul de $c\ a_1\ (c\ a_2\ (c\ a_3\ x))$. Il est à noter que certains langages de programmation manipulant des listes ont précisément en bibliothèque une fonction, usuellement appelée « fold » qui effectue ce calcul.

1. Coder les définitions précédentes en Coq (s'inspirer de la section 2.3 qui comporte des types paramétrés par d'autres types).
2. Donner en Coq la définition du calcul de la longueur d'une liste (rendre un pnat).
3. Donner en Coq la définition du calcul de la concaténation de deux listes.
4. *Bonus* Donner en Coq le calcul du renversement d'une liste.
5. *Bonus* Donner en Coq une fonction qui insère à sa place un entier dans une liste d'entiers croissante. Donner en Coq la fonction de tri par insertion.

2.6 Super bonus : arbres binaires et tri par arbre binaire de recherche

Suivant la même approche, les arbres binaires peuvent se représenter en λ-calcul par le type
$\texttt{arbin}\,\text{A} \stackrel{\text{def}}{=} \forall \text{T}, \text{T} \to (\text{T} \to \text{A} \to \text{T} \to \text{T}) \to \text{T}$.
Les constructeurs correspondants sont :

— l'arbre vide : $\texttt{pV}\,\text{A} : \texttt{arbin}\,\text{A} \stackrel{\text{def}}{=} \Lambda\text{T}.\lambda x^{\text{T}} c^{\text{T}\to\text{A}\to\text{T}\to\text{T}}.\ x$

— le nœud comprenant un sous-arbre gauche, un habitant de A et un sous-arbre droit
$\texttt{pN}\,\text{A} : \texttt{arbin}\,\text{A} \stackrel{\text{def}}{=} \lambda g^{\texttt{arbin}\,\text{A}} a^{\text{A}} d^{\texttt{arbin}\,\text{A}}.\Lambda\text{T}.\lambda x^{\text{T}} c^{\text{T}\to\text{A}\to\text{T}\to\text{T}}.\ c\ (g\ \text{T}\ x\ c)\ a\ (d\ \text{T}\ x\ c)$

1. Coder les définitions précédentes en Coq.
2. Coder en Coq le calcul de la liste des éléments d'un arbre binaire de la gauche vers la droite.
3. Coder en Coq l'insertion d'un entier dans un arbre binaire de recherche.
4. Coder en Coq un programme de tri de listes d'entiers utilisant les deux fonctions précédentes.
5. Coder en Coq une fonction qui vérifie si un arbre binaire est un arbre binaire de recherche.