Modèles de Calcul [Lambda-Calcul]
Programmation en λ-calcul (entiers)

Pascal Fradet   Jean-François Monin   Catherine Parent-Vigouroux

1 Codage des entiers de Church avec typage simple

Nous allons étudier le codage des entiers naturels en lambda-calcul inventé par Alonzo Church. Sa définition en lambda-calcul pur est la suivante :

— $c_0 \stackrel{\text{def}}{=} \lambda f x.x$
— $c_1 \stackrel{\text{def}}{=} \lambda f x.f\ x$
— $c_2 \stackrel{\text{def}}{=} \lambda f x.f\ (f\ x)$
— ...

Pour coder ces termes en Coq, il faut leur donner un type. On observe que ce sont des fonctions à deux paramètres f et x qui renvoient f appliqué itérativement n fois à x, n étant l'entier ainsi codé. Un type simple possible en Coq est donc :

```
Variable T: Set.
Definition cnat := (T->T) -> T->T.
```

Remarquer qu'une autre écriture possible de `cnat` est possible :

```
Definition cnat := (T->T) -> (T->T).
```

Cela revient à considérer un entier de Church comme une fonction qui prend en argument une fonction f et rend f itérée un certain nombre de fois, c'est-à-dire $f \circ f \ldots \circ f$.

1. Coder en Coq le type `cnat` et les 3 premiers entiers c_0, c_1 et c_2.
2. On définit la fonction successeur d'un entier de Church par le lambda-terme $\lambda n.\lambda f x.(f\ (n\ f\ x))$.
   Coder cette fonction en Coq, puis l'évaluer pour quelques entiers de Church.
3. Pour éviter d'avoir à introduire une constante c_n pour chaque n dont on aurait besoin, on choisit de définir une notation simplifiée pour le terme $\lambda f x.f\ (f\ \ldots x)$ où f est appliquée n fois. Pour cela, on a besoin de définir la composition de deux fonctions et un itérateur de composition.

```
(* Definition de la composition de g et f *)
Definition compo : (T->T) -> (T->T) -> (T->T) :=
  fun g f => fun x => g (f x).
(* Un raccourci syntaxique pour ecrire g ° f au lieu de (compo g f ) *)
Notation "g ° f" := (compo g f) (at level 10).
(* Un iterateur de f, n fois *)
Fixpoint iter (f:T->T) (n: nat) :=
  match n with
  | O => fun x => x
  | S p => f ° (iter f p)
  end.
(* Utilisation de cet itérateur pour construire un cnat a partir d'un nat standard *)
```

```
Definition cnat_of : nat -> cnat := fun n => fun f => (iter f n).
(* Raccourci syntaxique pour écrire le nᵉ entier de Church [n]N au lieu de (cnat n) *)
Notation "[ X ]N " := (cnat_of X) (at level 5).
(* par exemple [3]N signifie (cnat 3) et donc (après réduction) λ f x. f (f (f x))  *)
```

Coder ces différents éléments en Coq, tester le raccourci syntaxique pour les entiers de Church puis essayer la fonction successeur sur des entiers plus grands que 3.

4. (*) Démontrer
   `Remark cS_compo : forall n, cS n = fun f => f ° (n f)` et
   `Theorem cS_is_successor : forall n, cS [n]N = [S n]N.`

2 Opérations sur les entiers de Church avec typage simple

1. L'addition de deux entiers de Church est définie par : $\lambda n\lambda m.\lambda fx.(n\ f\ (m\ f\ x))$.
   Coder en Coq cette fonction, la valider sur des exemples.
2. (*) Énoncer et démontrer l'associativité de l'addition.
3. La multiplication de deux entiers de Church est définie par : $\lambda n\lambda m.\lambda f.(n\ (m\ f))$.
   Coder en Coq cette fonction et la tester sur des exemples.
4. Le test à zéro d'un entier de Church est défini par : $\lambda n.\lambda xy.(n\ (\lambda z.y)\ x)$.
   Coder en Coq cette fonction `tz` en utilisant le type `cbool` des booléens de Church comme type résultat de cette fonction de test à zéro.
5. (*) Prouver que `tz` renvoie `ctr` pour l'entier de Church `[0]N` et `cfa` pour n'importe quel autre entier de Church.