UFR IM²AG
UNIVERSITÉ Grenoble Alpes

# Modèles de Calcul [ Lambda-Calcul ]
# Programmation en λ-calcul pur et non typé

(booléens, entiers, couples, injections & récursion)

Martin Bodin, Pascal Fradet, Jean-François Monin

## 1 Codage des booléens et de la conditionnelle

Les booléens sont définis par `ctr` $\stackrel{\text{def}}{=} \lambda xy.x$ (pour *vrai*) et `cfa` $\stackrel{\text{def}}{=} \lambda xy.y$ (pour *faux*). L'opérateur conditionnel est codé par `cif` $\stackrel{\text{def}}{=} \lambda bmn.b\ m\ n$. Par exemple, `if` $b$ `then` $m$ `else` $n$ est encodé par `cif` $b\ m\ n$ (ou après simplification ($b\ m\ n$)). En effet on vérifie facilement que :

— `cif ctr` E F se simplifie (après 5 β-réductions) en E
— `cif cfa` E F se simplifie (après 5 β-réductions) en F.

Nous allons étudier et jouer avec ce codage des booléens avec les λ-expressions `lexp` en Coq. On les appellera les booléens de Church.

1. Coder en Coq (comme des `lexp`), les booléens et la conditionnelle.
   `Definition ctr := ...`, `Definition cfa := ...`, `Definition cif := ...`
2. Vérifier (en utilisant `show_cbn`) que l'évaluation de `cif` sur vrai (`ctr`) et sur faux (`cfa`) est correcte.

## 2 Codage des opérateurs booléens

Regardons comment coder la négation sur les booléens de Church. La négation de $b$ peut être construite de deux manières. La première, en indiquant que le résultat représente faux si $b$ est vrai et qu'il représente vrai si $b$ est faux, ce qui se code par $b$ `cfa ctr`. Cette fonction de négation est donc $\lambda b.b\ (\lambda xy.y)\ (\lambda xy.x)$. La seconde manière consiste à « factoriser $\lambda xy.$ » dans le code précédent, c'est-à-dire à écrire le résultat sous la forme $\lambda xy.\mathrm{U}_{b,x,y}$, où $\mathrm{U}_{b,x,y}$ est un terme à trouver qui produit respectivement $y$ si $b$ est vrai (et l'expression globale devient $\lambda xy.y$ qui représente faux) et $x$ si $b$ est faux (et l'expression globale devient $\lambda xy.x$ qui représente vrai). Autrement dit, $\mathrm{U}_{b,x,y}$ peut s'écrire `if` $b$ `then` $y$ `else` $x$, ce qui se code $b\ y\ x$. La négation de $b$ est donc $\lambda xy.b\ y\ x$, et la fonction de négation correspondante est $\lambda b.\lambda xy.b\ y\ x$.

1. On veut maintenant coder les opérateurs booléens `and` et `or`, en commençant par les versions « factorisées ». Compléter les définitions suivantes :
   — `cand` $\stackrel{\text{def}}{=} \lambda ab.\lambda xy.a\ (b\ \square\ \square)\ \square$
   — `cor` $\stackrel{\text{def}}{=} \lambda ab.\lambda xy.\square\ x\ (\square\ x\ y)$,
   — `cor_cif` $\stackrel{\text{def}}{=} \lambda ab.\lambda xy.$`cif` $\square\ x\ (\square\ x\ y)$
   — `cor'` $\stackrel{\text{def}}{=} \lambda a.a\ \square$ (version donnée en cours, dérivée à partir de la version non factorisée)
2. Coder `cnot` (version factorisée), `cnot'` (version non factorisée), `cand`, `cor` et `cor'` en Coq comme des `lexp`.
3. Vérifier avec `red_cbn` ou `show_cbn` les tables de vérité de ces opérateurs. Par exemple (`cnot ctr`) se réduit bien en `cfa`, (`cnot cfa`) se réduit bien en `ctr`; (`cand ctr cfa`) se réduit bien en `cfa`, etc.
4. Observer ce qui se passe avec d'autres stratégies de réduction, notamment les stratégies faibles (`show_wcbn`). Il est intéressant de regarder `cor_cif` puis, par exemple `(cor_cif cfa ctr)` .

## 3 Codage des entiers

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

— $c0 \stackrel{\text{def}}{=} \lambda f x.x$
— $c1 \stackrel{\text{def}}{=} \lambda f x.f\ x$
— $c2 \stackrel{\text{def}}{=} \lambda f x.f\ (f\ x)$
— $cn \stackrel{\text{def}}{=} \lambda f x.\underbrace{f\,(\ldots(f}_{n}\ x)\ldots)$

1. Coder en Coq (comme des `lexp`) les 4 premiers entiers $c_0$, $c_1$, $c_2$ et $c_3$.
2. La fonction successeur consiste à ajouter un $f$ au codage de l'entier passé en paramètre. Compléter la définition de la fonction successeur d'un entier de Church :
   `csucc` $\stackrel{\text{def}}{=} \lambda n.\lambda f x.\square\ (n\ \square\ \square)$.
   Coder cette fonction en Coq, puis l'évaluer pour quelques entiers de Church.

## 4 Opérations sur les entiers

1. La fonction addition de deux entiers de Church consiste à "concaténer" leur réprésentation. Compléter la définition de l'addition :

   `cadd` $\stackrel{\text{def}}{=} \lambda n m.\lambda f x.\ n\ \square\ (m\ \square\ \square)$.

   Coder en Coq (comme une `lexp`) cette fonction et la tester sur des exemples (entiers de Church, mais essayer aussi des booléens!).
2. La fonction multiplication de deux entiers de Church $n$ et $m$ consiste à se servir du codage du premier pour dupliquer $n$ fois la représentation du deuxième (et obtenir $n * m$ occurrences de $f$). Compléter la définition de la multiplication :

   `cmult` $\stackrel{\text{def}}{=} \lambda n m.\lambda f.\ \square\ (\square\ \square)$.

   Coder en Coq (comme une `lexp`) cette fonction et la tester sur des exemples.
3. On peut également définir des fonctions utilisant à la fois le codage des entiers et booléens. Compléter la définition du test à zéro d'un entier de Church :

   `ceq0` $\stackrel{\text{def}}{=} \lambda n.\lambda x y.n\ (\lambda z.\square)\ \square$.

   qui rend `ctr` si son argument $n$ est `c0` et rend `cfa` sinon.

   Coder en Coq (comme une `lexp`) cette fonction et la tester sur des exemples.

## 5 Structures de données

**A)** Un couple $(x_1, x_2)$ est représenté en lambda-calcul par la fonction $\lambda k.k\ x_1\ x_2$. La fonction `cpl` prend deux arguments et construit un couple :

`cpl` $\stackrel{\text{def}}{=} \lambda x_1 x_2.\lambda k.\ k\ x_1\ x_2$

Pour accéder aux éléments d'un couple il suffit de lui passer une fonction dite de *continuation*. Ainsi, $(\lambda k.k\ x_1\ x_2)\ f$ se réduit en $f\ x_1\ x_2$ et la continuation $f$ peut manipuler les deux éléments du couple.

1. Coder en Coq (comme des `lexp`) les continuations `kfst` et `ksnd` qui, passées en argument à un couple, retournent respectivement son premier et son deuxième élément. Tester sur le couple ($\lambda k.k$ `ctr cfa`). Coder ensuite les opérateurs `fst` et `snd` qui retournent respectivement le premier et le deuxième éléments d'un couple. Tester sur `(cpl ctr cfa)`.

2. Coder en Coq une fonction prenant en argument un couple $c$ d'entiers et qui rend, en passant à $c$ une continuation appropriée, la somme des deux composantes de $c$.

**B)** Dualement, on peut concevoir une structure de données représentant non pas la juxtaposition mais le choix entre deux données. Le principe est semblable à celui des couples, sauf qu'au lieu d'*une* fonction de formation (de couple) on en a deux, appelées *injections* qui, à partir d'un argument $x$ rendent respectivement $\lambda k_1 k_2.k_1\ x$ et $\lambda k_1 k_2.k_2\ x$. Le résultat de chacune de ces injections prend donc en argument *deux* continuations à *un* argument (au lieu d'*une* continuation à *deux* arguments pour le couple). Pour accéder une telle donnée on lui passe deux continuations. Ainsi, $(\lambda k_1 k_2.k_1\ x)\ f\ g$ se réduit en $f\ x$ tandis que $(\lambda k_1 k_2.k_2\ x)\ f\ g$ se réduit en $g\ x$.

1. Coder en Coq les injections `inj1` et `inj2`.
2. Coder en Coq une fonction prenant en argument une donnée qui est soit un entier $n$ (emballé par `inj1`) soit un booléen $b$ (emballé par `inj2`) et qui rend le double de $n$ dans le premier cas, la négation de $b$ dans le second. Tester.
3. La *donnée optionnelle* correspond à un choix entre une donnée (`Some` $x$) ou son absence (`None`). Elle se représente avec une continuation à un argument et une continuation à zéro argument. Coder en Coq les injections `Some` et `None`, puis une fonction `osucc` prenant en argument un entier optionnel (`Some` $n$ ou `None`) et rendant un entier optionnel qui est `Some` $(n+1)$ dans le premier cas et `Some` 0 dans le second. Tester.

## 6 Codage du prédécesseur

Pour définir le prédécesseur sur les entiers de Church, on utilise les couples. L'idée est d'itérer $n$ fois une fonction agissant sur des couples d'entiers. Prenons une fonction $g$ qui à partir de $(x,y)$ donné en argument rend $(y,y+1)$. Alors, en itérant $n$ fois $g$ sur $(0,0)$, on obtient $(n-1,n)$ et il suffit d'extraire la première composante de ce couple. Si $n$ est codé comme un entier de Church alors itérer $n$ fois une fonction $g$ sur un argument $a$ est juste $(n\ g\ a)$. En effet,

$$(\lambda f x.\underbrace{f\,(\ldots(f}_{n}\ x)\ldots)\ g\ a \xrightarrow{\beta} \underbrace{g\,(\ldots(g}_{n}\ a)\ldots)$$

1. Coder en Coq (comme une `lexp`) la fonction `iter` qui prend un entier de Church $n$, une fonction $g$ et un argument $x$ et qui applique $n$ fois la fonction $g$ sur $x$. La tester avec la fonction `csucc` et `c0`.
2. Coder en Coq (comme une `lexp`) la fonction `cpred1` qui à partir d'un couple $\lambda k.\,k\ x\ y$ donné en argument rend le couple $\lambda k.\,k\ y$ (`csucc` $y$). Tester `cpred1` sur le couple $\lambda k.\,k\ c1\ c2$.
3. Compléter la définition du prédécesseur d'un entier de Church :

   `cpred` $\stackrel{\text{def}}{=}$ $\lambda n.$`fst` (`iter` $\square\,\square\,\square$) ou plus simplement `cpred` $\stackrel{\text{def}}{=}$ $\lambda n.$`fst` $(\square\,\square\,\square)$.

   Coder en Coq comme une `lexp` cette fonction et la tester sur quelques exemples.
4. Définir une autre version du prédécesseur en utilisant `osucc` et une valeur initiale convenable :

   `cpredo` $\stackrel{\text{def}}{=}$ $\lambda n.\square\,\square\,\square\ (\lambda x.\,x)\ c_0$. Tester.

## 7 Combinateur de point fixe

Posons $\Delta \stackrel{\text{def}}{=} \lambda f.\lambda x.\,f(x\,x)$, alors on peut définir le combinateur de point fixe de Curry par :

$\mathbf{Y} \stackrel{\text{def}}{=} \lambda f.(\Delta f)\,(\Delta f))$. Pour un $\lambda$-terme U quelconque on a la suite de réductions

$$\mathbf{Y}\,\mathrm{U} \xrightarrow{\beta} (\Delta\,\mathrm{U})\,(\Delta\,\mathrm{U}) \xrightarrow{\beta} (\lambda x.\mathrm{U}\,(x\,x))\,(\Delta\,\mathrm{U}) \xrightarrow{\beta} \mathrm{U}\,((\Delta\,\mathrm{U})\,(\Delta\,\mathrm{U}))$$

1. Donner un exemple de U pour lesquel ($\mathbf{Y}$ U) n'a pas de forme normale et un exemple de U pour lesquel ($\mathbf{Y}$ U) a une forme normale (les solutions les plus simples prennent 4 caractères).
2. Définir $\mathbf{Y}$ en Coq comme une `lexp`.

**Remarque.** La raison pour laquelle **Y** est appelé combinateur de point fixe est la suivante. Si on considère que deux λ-termes U et V sont égaux si l'on peut passer de l'un à l'autre par des β-réductions, dans un sens ou dans l'autre [1] les questions précédentes nous amènent à conclure $\mathrm{U}(\mathbf{Y}\,\mathrm{U}) = \mathbf{Y}\,\mathrm{U}$. Autrement dit, $\mathbf{Y}\,\mathrm{U}$ est un point fixe de U, c'est-à-dire une solution de l'équation $\mathrm{U}(x) = x$.

## 8 Codage de définitions récursives par point fixe

Illustrons la notion de *définition récursive* par l'exemple bien connu de la fonction factorielle :

$$\mathsf{fact} \;=\; \lambda n.\,\textbf{if}\;\; n = 0\;\; \textbf{then}\;\; 1\;\; \textbf{else}\;\; n \times \mathsf{fact}\,(n-1) \tag{1}$$

Affirmer qu'il s'agit là d'une définition serait une escroquerie car on utiliserait fact pour définir fact. Pour échapper à ce problème on définit d'abord la fonction

$$\mathbf{F} \;\stackrel{\text{def}}{=}\; \lambda f.\lambda n.\,\textbf{if}\;\; n = 0\;\; \textbf{then}\;\; 1\;\; \textbf{else}\;\; n \times f\;(n-1) \tag{2}$$

Cette fonction **F** est appelée la *fonctionnelle*[2] associée à la définition récursive fact. Une fois que cette fonctionnelle **F** est posée, l'équation récursive (1) se comprend comme l'affirmation suivante :

$$\mathsf{fact} \;=\; \mathbf{F}\,(\mathsf{fact}) \tag{3}$$

Autrement dit, comme l'affirmation que fact est un point fixe de **F**, c'est-à-dire une solution de l'équation

$$x \;=\; \mathbf{F}(x) \tag{4}$$

Pour que cela puisse être accepté comme une définition de fact, encore faut-il s'assurer

— que l'équation (4) a bien une solution ;

— que cette solution est unique.

Cela n'a rien d'évident a priori. Oublions un instant les points fixes sur les fonctions et considérons des points fixes sur des objet plus simples, les entiers. Par exemple, les équations suivantes :

$$x = x+1 \qquad x = 2-x \qquad x = x*x \qquad x = x$$

ont respectivement 0, 1, 2 et une infinité de solutions.

Le même genre de phénomène est susceptible d'arriver sur les points fixes de fonctions. Moyennant certaines conditions sur les fonctionnelles, on peut garantir l'existence et l'unicité d'un tel point fixe [3].

Pour revenir au λ-calcul, nous avons vu à la section précédente que le combinateur **Y** permet précisément de calculer un point fixe d'une fonction U. Nous avons tous les ingrédients (entiers, booléens, conditionnelle, opérateurs, point fixe) pour définir des programmes (fonctions) sur les entiers ou booléens. On va maintenant définir la fonction factorielle en λ-calcul pur et non typé.

1. En utilisant les opérateurs définis précédemment (`cif`, `ceq0`, `cmult`, `cpred`), définir en Coq comme une `lexp` la fonctionnelle associée à fact.

   ```
   Definition cfonc := \f n· ...
   ```

2. Définir la fonction factorielle en Coq comme une `lexp`

   ```
   Definition cfact :=  ...
   ```

   et la tester avec `red_cbn` (sur de tout petits entiers < 4).

1. Mathématiquement on dira qu'ils sont dans la même classe d'équivalence de la relation obtenue par clôture réflexive, symétrique et transitive de $\xrightarrow{\beta}$.

2. Une fonctionnelle est une fonction de fonctions.

3. Ou plus exactement, celles d'un *plus petit* point fixe. La relation de comparaison entre fonctions utilisée est : $f \leq g$ ssi $f$ est une approximation de $g$ ; plus précisément : pour tout $x$, si le calcul de $f\,x$ termine et produit une valeur, alors $g\,x$ termine et produit la même valeur – $f\,x$ pourrait ne pas terminer là où $g\,x$ le pourrait. On ne détaille pas davantage ici.