# Introduction à la sémantique dénotationnelle

Cassis
Juillet 2022

## Introduction

Ce document cherche à introduire la sémantique dénotationnelle. Il s'adresse en priorité à des néophytes sur le sujet qui ont malgré tout de l'expérience en mathématiques. Nous utiliserons comme modèle le langage OCaml. Les parties seront réparties d'abord en une présentation de ce que nous allons utiliser dans OCaml (nous utiliserons seulement une version réduite de ce langage), puis nous parlerons de domaines et d'ordre partiel complet, en utilisant comme exemple notre modèle de OCaml.

# Table des matières

# Première partie
# Une description de OCaml

Cette partie va décortiquer la sémantique dite opérationnelle de OCaml, qui sera donc la donnée des termes du langage ainsi que des règles pour effectuer un calcul dans le langage (on appelle cela une réduction). Pour ce faire, nous présenterons tout d'abord le lambda-calcul de façon succincte, avant de donner la définition du lambda-calcul simplement typé, sur lequel nous ajouterons enfin des constructions propres à OCaml (telles que le `let x = e in e'`).

## 1 Le lambda-calcul

### 1.1 Définition du lambda-calcul

Le lambda-calcul, à l'origine inventé par Church dans les années 1920 pour définir la notion de calculabilité, a ensuite permis l'essor de la programmation fonctionnelle. Le principe est simple : tout ce que l'on manipule sont des fonctions. Les objets manipulés seront appelés lambda-termes, et nous auront pour les définir besoin d'un ensemble $\mathcal{V}$ infini de variables, que nous noterons avec des lettres $(x, y, z, a, b, \ldots)$. Nous allons maintenant définir l'ensemble $\Lambda$ des lambda-termes.

**Définition 1.1** (Lambda-terme)**.** On appelle lambda-terme un élément $M \in \Lambda$ construit par la grammaire suivante :

$$M, N ::= x \mid \lambda x.M \mid (MN)$$

Cette grammaire est donnée sous une forme de Backus-Naur. Cette forme consiste à donner une définition récursive en explicitant comment construire les éléments de base (ici les variables symbolisées par le $x$) et comment, à partir d'une expression, en construire une plus grand (ici les deux autres cas). Nous utiliserons abondamment cette notation.

Décrivons rapidement ce que signifie chaque partie de notre définition :

1. Un terme de la forme $x$ n'a pas de signification propre, il en aura par rapport à un terme plus grand.
2. Un terme de la forme $\lambda x.M$ correspond à la fonction $x \mapsto M$, qui à $x$ associe l'expression $M$. $x$ apparaîtra en général dans $M$, mais ça n'est pas forcément le cas.
3. Un terme de la forme $(MN)$ correspond à l'application de la fonction qui correspond à $M$ à l'argument qui correspond à $N$. On remarque donc qu'une expression de la forme $(\lambda x.M)N$ est une expression que l'on pourrait lire « l'expression $M$ où $x$ est remplacé par $N$ ». Nous verrons cela plus tard. Une dernière remarque : on écrit largement $f\ x$ au lieu de $f(x)$ pour simplifier nos notations.

**Exemple 1.1.** Donnons quelques lambda-termes classiques :

- $I = \lambda x.x$ correspond à la fonction identité : $I : x \mapsto x$.
- $\top = \lambda x.\lambda y.x$ correspond à la fonction donnant une fonction constante. On peut aussi la voir comme une fonction de deux variables, $x$ et $y$, retournant la première variable $x$.
- $\bot = \lambda x.\lambda y.y$ correspond à la fonction de deux variables qui retourne la deuxième variable.
- $\Omega = \omega\omega = (\lambda x.xx)(\lambda x.xx)$ représente un calcul qui ne se termine pas. On verra en effet que $\Omega$ donne un calcul infini valant $\Omega$ à chaque étape.

Faisons ici un *aparte* pour détailler ce que nous entendons par fonction à plusieurs variables. En effet, le lambda-calcul ne permet de manipuler que des fonctions à une variable et ne permet pas de manipuler des couples en premier lieu. Cependant, une fonction $(x, y) \mapsto f(x, y)$ est strictement équivalente à une fonction $x \mapsto (y \mapsto f(x, y))$, c'est ce qu'on appelle la curryfication. Nous allons donc considérer comme fonction à plusieurs variables une fonction de cette forme. Pour faciliter la lecture, nous emploierons alors le raccourci $\lambda xy.$ pour signifier $\lambda x.\lambda y.$ et ainsi de suite pour un nombre de variables quelconques. Enfin, et encore pour faciliter la lecture, nous allons supposer que l'application

de fonction est associative à gauche, ce qui signifie que $f\ x\ y$ se lit $(f\ x)\ y$, nous permettant de manipuler les fonctions à plusieurs variables en écrivant juste les variables à la suite. Par exemple, avec ces notations, $\top\ 1\ 0 = 1$.

**Exercice 1.1.** Pour cet exercice, on suppose qu'on dispose quand même des entiers naturels et des opérations arithmétiques. Écrire un lambda-terme correspondant :

- à la fonction $x \mapsto x + 1$
- à la fonction $(x, y) \mapsto x + y$
- à l'expression $((x, y) \mapsto x + y)\ 13\ 12$
- à la composition : $\circ : (f, g) \mapsto (x \mapsto f\ g\ x)$

### 1.1.1 Variables libres, liées, alpha-équivalence

Remarquons que les variables à l'intérieur d'une fonction sont muette, c'est-à-dire que la fonction $\lambda y.y$ est moralement la même que $I = \lambda x.x$ précédemment définie. Il convient donc de définir une notion permettant de traiter les variables muettes ou non. Cette notion est celle de variable libre et de variable liée. Une variable libre est une variable qui est fixée, qui n'apparaît pas dans un $\lambda$ placé « plus haut » dans l'expression. Une variable liée, au contraire, est une variable telle qu'on peut trouver un $\lambda$ qui la rend muette.

Nous allons donner une définition formelle de ces notions sous forme de règles d'inférences. Ces règles permettent de définir une relation par induction en énonçant les prémisses de la relation en haut d'une barre horizontale et sa conclusion en bas.

**Définition 1.2** (Variable libre, variable liée, formule close)**.** Définissons l'ensemble $\mathrm{vl}(M)$ des variables libres de $M$ par induction sur la structure de $M$ :

$$\frac{}{\mathrm{vl}(x) = \{x\}} \qquad \frac{\mathrm{vl}(M) = A}{\mathrm{vl}(\lambda x.M) = A \setminus \{x\}} \qquad \frac{\mathrm{vl}(M) = A \quad \mathrm{vl}(N) = B}{\mathrm{vl}(MN) = A \cup B}$$

On définit l'ensemble des variables liées par induction, de la même façon :

$$\frac{}{\mathrm{vlie}(x) = \varnothing} \qquad \frac{\mathrm{vlie}(M) = A}{\mathrm{vlie}(\lambda x.M) = A \cup \{x\}} \qquad \frac{\mathrm{vlie}(M) = A \quad \mathrm{vlie}(N) = B}{\mathrm{vlie}(MN) = A \cup B}$$

On appelle formule close un lambda-terme $M$ tel que $\mathrm{vl}(M) = \varnothing$. Nous utiliserons principalement des formules closes.

Enfin, on appelle ensemble des variables d'une expression $M$, et l'on note v, l'ensemble

$$\mathrm{v}(M) = \mathrm{vl}(M) \cup \mathrm{vlie}(M)$$

**Exercice 1.2.** Donner les variables libres des expression suivantes :

- I
- $\lambda y.x$
- $(\lambda x.yy)(\lambda y.zz)$

Ces notions permettent de définir rigoureusement comment renommer une variable. Pour cela, il faut faire attention au cas de la capture de variable. Par exemple si l'on prend $\lambda x.y$ et que l'on renomme $x$ en $y$, alors on obtient $\lambda y.y$ : ceci n'est pas satisfaisant puisque les deux termes n'ont pas du tout le même sens. Pour éviter ce problème, on va faire, chaque fois qu'il y a une lambda-abstraction (un terme de la forme $\lambda x.M$) avec une variable libre du même nom que celle que l'on souhaite renommer, un changement de nom de la variable du $\lambda$.

**Définition 1.3** (Substitution)**.** Soit $M$ et $N$ deux lambda-termes. On note $M[N/x]$ le lambda-terme défini par :

$$\frac{}{x[N/x] = N} \qquad \frac{}{y[N/x] = y} \qquad \frac{M[N/x] = M' \quad P[N/x] = P'}{(MP)[N/x] = M'P'}$$

$$\frac{}{(\lambda x.M)[N/x] = \lambda x.M} \qquad \frac{M[z/y][N/x] = M' \quad z \notin \mathrm{vl}(N) \cup \mathrm{vl}(M)}{(\lambda y.M)[N/x] = \lambda z.M'}\ y \in \mathrm{vl}(N) \qquad \frac{M[N/x] = M'}{(\lambda y.M)[N/x] = \lambda y.M'}$$

Le terme à côté de la règle d'inférence numéro 5 donne une condition supplémentaire d'application de la règle.

**Exemple 1.2.** Voici plusieurs substitutions :

- $I[y/x] = \lambda y.y$
- $(\lambda x.x\ y)[a/y] = \lambda x.x\ a$
- $(\lambda x.x\ y)[x/y] = \lambda z.z\ x$

Nous nommons alpha-conversion ou alpha-équivalence le fait que deux termes qui ne varient qu'au nom près de leurs variables liées sont des termes identiques.

**Définition 1.4** (Alpha-équivalence)**.** Soient $M$ et $N$ deux lambda-termes, on définit la relation $=_\alpha$ par les règles :

$$\frac{}{M =_\alpha M} \qquad \frac{M =_\alpha N \quad N =_\alpha P}{M =_\alpha P} \qquad \frac{M =_\alpha N}{N =_\alpha M}$$

$$\frac{}{\lambda x.M =_\alpha \lambda y.M[y/x]} \qquad \frac{M =_\alpha M' \quad N =_\alpha N'}{(M\ N) =_\alpha (M'\ N')} \qquad \frac{M =_\alpha N}{\lambda x.M =_\alpha \lambda x.N}$$

*Remarque.* La première ligne permet de définir l'alpha-équivalence comme une relation d'équivalence. La deuxième ligne possède essentiellement la première règle, disant qu'un renommage dans un terme ne change pas le terme. Les autres règles de la deuxième ligne expriment que l'on peut effectuer ce renommage « à l'intérieur » des termes.

Nous travaillerons désormais non pas dans $\Lambda$ mais dans $\Lambda/_{=_\alpha}$ pour assimiler des lambda-termes alpha-équivalents (nous utiliserons donc désormais $\Lambda$ pour dire $\Lambda/_{=_\alpha}$).

### 1.1.2 Réduction de termes

La réduction est l'élément central du lambda-calcul. En effet, c'est par lui que l'on obtient la sémantique opérationnelle du lambda-calcul, c'est-à-dire le comportement des termes de notre système formel. On appelle ces réductions des beta-réductions. Elles correspondent exactement à l'intuition que nous avons donnée au début de cette section : l'application à un argument $N$ de la fonction $\lambda x.M$ se réduit en $M$ dans lequel on a remplacé les occurrences de $x$ par $N$ : c'est exactement $M[N/x]$.

**Définition 1.5** (Beta-réduction)**.** On définit la beta-réduction par les règles suivantes :

$$\frac{}{(\lambda x.M)N \to_\beta M[N/x]} \qquad \frac{M \to_\beta M'}{(\lambda x.M) \to_\beta (\lambda x.M')} \qquad \frac{M \to_\beta M'}{(MN) \to_\beta (M'N)} \qquad \frac{N \to_\beta N'}{(MN) \to_\beta (MN')}$$

On appelle redex, de l'anglais *reducible expression*, une expressoin de la forme $(\lambda x.M)N$.

On dit qu'un lambda-terme $M$ est sous une forme normale s'il n'existe pas de terme $N$ tel que $M \to_\beta N$.

On dit que $M$ se réduit en $N$ si $M \to_\beta^* N$, où $\to_\beta^*$ est la clôture transitive et réflexive de $\to_\beta$.

**Exercice 1.3.** On peut ainsi calculer les réductions suivantes :

- $II \to_\beta^* I$
- $\top I(\bot\top\top) \to_\beta^* I$
- $\Omega \to_\beta^* \Omega$

*Remarque.* Nous utilisons des notations telles que $\top$ ou $I$. Ces notations permettent de factoriser le code et donc d'éviter d'avoir des lignes incompréhensibles car surchargées de symboles. En effet, $II$ semble directement donner l'identité, rien que par l'habitude des mathématiques.

Nous allons enfin définir une équivalence à partir de cette relation, appelée beta-équivalence. Les termes beta-équivalents correspondent à des termes qui donnent le même résultat.

**Définition 1.6** (Beta-équivalence)**.** Deux termes $M$ et $N$ sont dits beta-équivalents, que l'on note $M =_\beta N$, s'ils sont équivalents pour la clôture symétrique, réflexive et transitive de $\to_\beta$. Ceci signifie qu'il existe une suite $M, M_1, \ldots, M_n, N$ (potentiellement nulle, où $M = N$) telle que $M_i \to_\beta M_{i+1}$ ou $M_i \leftarrow_\beta M_{i+1}$ (en considérant $M = M_0$ et $N = M_{n+1}$).

### 1.1.3 Propriétés du lambda-calcul

Nous citerons ici, sans les démontrer, les propriétés essentielles du lambda-calcul.

**Proposition 1.1.1** (Confluence)**.** *Soient $M, N, P \in \Lambda$ tels que $M \to_\beta N$ et $M \to_\beta P$, alors il existe un terme $Q$ tel que $N \to_\beta^* Q$ et $M \to_\beta^* Q$.*

Ce résultat de confluence nous indique que l'ordre dans lequel on effectue les réductions dans un terme importe peu.

Le prochain résultat, appelé la propriété de Church-Rosser, assure entre autre qu'il y a unicité, si elle existe, de la forme normale d'un terme (la fore normale à laquelle on aboutit en effectuant un maximum de réductions à la suite depuis le terme en question).

**Théorème 1.1** (Church-Rosser)**.** *Soit $(M, N) \in \Lambda^2$ tels que $M =_\beta N$. Alors il existe un terme $v$ tel que $M \to_\beta v$ et $N \to_\beta v$.*

Nous devons maintenant préciser la notion de terme normalisable et fortement normalisable. En effet, on remarque que le terme $\Omega$, se réduisant vers lui-même, n'arrive jamais à une forme normale : il y a donc des termes ne possédant pas de forme normale, et certains en possédant une.

**Définition 1.7** (Terme normalisable)**.** Un terme $M$ est dit normalisable s'il existe $N$ une forme normal telle que $M \to_\beta^* N$. Il est dit fortement normalisable si toute suite de réduction partant de $M$ est finie (cela équivaut à ce que toute suite de réduction converge vers une unique forme normale, par la propriété de Church-Rosser).

**Exemple 1.3.**

- Comme nous l'avons vu, $\Omega$ n'est pas normalisable.
- Le terme $\top\omega\omega$ (où $\omega = \lambda x.x\ x$) est fortement normalisable.
- le terme $(\lambda x.y)(\omega\omega)$, lui, est normalisable mais non fortement normalisable, puisqu'on peut effectuer une suite infinie de réductions sur $\omega\omega$.

**Exercice 1.4.** Nous avons vu un lambda-terme restant constant lors de sa réduction, $\Omega$. Trouver un lambda-terme dont la taille grandit à mesure de ses réductions.

**Exercice 1.5** $(*)$**.** Trouver un lambda-terme $M$ tel que :

1. $M$ est normalisable.
2. $M\ I$ n'est pas normalisable.
3. $M\ I\ I$ est normalisable.
4. $M\ I\ I\ I$ n'est pas normalisable.

avec la définition précédente de $I$.

## 1.2 Coder en lambda-calcul

Cette section définira le codage en lambda-calcul des entiers, des booléens, des couples et des opérations arithmétiques de base, avant de parler de récursion. Nous utiliserons pour cela l'égalité $=_\beta$ définie plus tôt, car cette égalité nous permet d'ignorer les détails de codage une fois que l'on a établi un codage (on travaillera ainsi, au fil de cette section, avec de plus en plus de boîtes noires).

### 1.2.1 Les entiers de Church

Nous allons maintenant voir comment définir un entier en lambda-calcul. Pour cela, l'idée est qu'un nombre $n$ peut se représenter en tant que fonction comme une itérée. Ceci signifie que le nombre $n$ sera représenté par la fonction $\overline{n} = \lambda f.\lambda x.f^n\ x$ où $f^0\ x = x$ et $f^{n+1}\ x = f^n\ (f\ x)$. De plus, on définit les entiers à partir de deux primitives (ou constructeurs) : l'élément 0 et l'élément $S$, ou successeur, qui correspond à la fonction $x \mapsto x + 1$.

**Définition 1.8** (Entier de Church)**.** On définit les entiers de Church par les constructeurs suivants :

$$\overline{0} = \lambda f.\lambda x.x \qquad S\ \overline{n} = \lambda f.\lambda x.\overline{n}\ f\ (f\ x)$$

**Exemple 1.4.** Calculons ensemble $\overline{1}$ et $\overline{2}$ :

$$\begin{aligned}\overline{1} &= \lambda f.\lambda x.\overline{0}\ f\ (f\ x)\\ &= \lambda f.\lambda x.(\lambda f.\lambda x.x)\ f\ (f\ x)\\ &\to_\beta \lambda f.\lambda x.f\ x\end{aligned}$$

$$\begin{aligned}\overline{2} &= \lambda f.\lambda x.\overline{1}\ f\ (f\ x)\\ &\to_\beta \lambda f.\lambda x.f\ (f\ x)\\ &= \lambda f.\lambda x.f^2\ x\end{aligned}$$

Ainsi nos définitions semblent bien nous donner que $\overline{n} =_\beta \lambda f.\lambda x.f^n\ x$.

**Exercice 1.6.** Montrer par récurrence que $S^n\ \overline{0} \to_\beta^* \lambda f.\lambda x.f^n\ x$

*Remarque.* Le coodage des entiers de Church nous donne naturellement un principe d'itération : si l'on veut appliquer $n$ fois la fonction $f$ à l'argument $x$, alors il suffit simplement d'écrire $\overline{n}\ f\ x$.

Nous avons maintenant deux choix pour coder l'addition. La première, directe, est

$$\text{add} = \lambda n.\lambda m.\lambda f.\lambda x.n\ f\ (m\ f\ x)$$

mais nous utiliserons plutôt une définition utilisant $S$ :

**Définition 1.9** (Addition)**.** Nous définissons l'addition comme le lambda-terme suivant :

$$\text{add} = \lambda m.\lambda n.n\ S\ m$$

et nous noterons $a + b$ pour add $a$ $b$.

*Remarque.* Dans les deux définition, il n'y a pas de barre sur $n$ et $m$. Cela est dû au fait que dedans, $m$ et $n$ sont juste des fonctions quelconques (l'argument de add pourrait très bien être $\top$) et non les fonctions particulières de la forme $\overline{n}$.

**Exercice 1.7.** Montrer par récurrence que les deux définitions de add retournent bien la somme de deux nombres, c'est-à-dire que $\overline{n} + \overline{m} = \overline{n+m}$.

**Définition 1.10** (Multiplication)**.** Nous définissons la multiplication comme le lambda-terme suivant :

$$\text{mult} = \lambda n.\lambda m.\lambda f\ x.n\ (m\ f)\ x$$

et nous noterons $a \times b$ pour mult $a$ $b$.

**Exercice 1.8.** Montrer par récurrence que $\overline{n} \times \overline{m} = \overline{n \times m}$.

### 1.2.2 Les booléens et les couples

Nous avons en fait déjà défini les booléens, c'est-à-dire les deux valeurs $\top$ et $\bot$. Ce sont donc les deux projections de $\Lambda^2$ sur $\Lambda$ curryfiées. Ces deux valeurs permettent de coder l'expression `if` $b$ `then` $e$ `else` $e'$ pour $b$, $e$ et $e'$ trois expressions, dont $b$ représente un booléen.

**Définition 1.11** (If/then/else)**.** On définit `if` $b$ `then` $e$ `else` $e'$ par l'expression :

$$\texttt{if}\ b\ \texttt{then}\ e\ \texttt{else}\ e' = b\ e\ e'$$

On voit bien, alors, que

$$\texttt{if } \top \texttt{ then } e \texttt{ else } e' \to_\beta e$$

et

$$\texttt{if } \bot \texttt{ then } e \texttt{ else } e' \to_\beta e'$$

On peut aussi définir les opérateurs booléens classiques.

**Définition 1.12** (Opérateurs booléens)**.** Les opérateurs $\vee$, $\wedge$ et $\neg$ sont définis par leur fonction curryfiée, préfixe, suivante :

$$\text{or} = \lambda b.\lambda b'.b \top b' \qquad \text{and} = \lambda b.\lambda b'.b\ b' \bot \qquad \text{not} = \lambda b.b \bot \top$$

**Exercice 1.9.** Montrer que ces codages sont corrects, par disjonction de cas quand $b$ et $b'$ prennent des valeurs dans $\{\top, \bot\}$.

Concentrons-nous maintenant sur le codage des couples en lambda-calcul. Un couple est caractérisée par les deux éléments qui le constituent : on peut donc la caractériser par le résultat de ses projections. Un couple $\langle a, b\rangle$ peut donc s'écrire par le couple $\langle \pi_1\langle a, b\rangle, \pi_2\langle a, b\rangle\rangle$. Si cette écriture complexifie tout, on peut maintenant curryfier l'intérieur de chaque argument pour obtenir $\langle \top\ a\ b, \bot\ a\ b\rangle$. Ainsi, un couple est caractérisée par le résultat qu'elle a lorsqu'on y applique $\top$ ou $\bot$.

**Définition 1.13** (Couple)**.** On définit $\langle x, y\rangle$, le couple constituée de $x$ et $y$, par le lambda-terme :

$$\langle x, y\rangle = \lambda b.b\ x\ y$$

On note, de plus, $\pi_1 = \top$ et $\pi_2 = \bot$. Si ces notations semblent être inutiles puisque nous avons déjà une notation pour $\top$ et $\bot$, les expressions identiques expriment des utilisations différentes et améliorent la lisibilité (d'ailleurs $\overline{0}$ est aussi un terme identique à $\bot$).

**Exercice 1.10.** Trouver un lambda-terme représentant une fonction `eq0` qui renvoie $\top$ à $\overline{0}$ et $\bot$ à $\overline{n}$ pour $n > 0$.

**Exercice 1.11** (L'opération prédécesseur)**.** Cet exercice vise à définir la fonction pred : $x \mapsto x - 1$, définie pour les entiers en convenant que $\text{pred}(0) = 0$.

1. Écrire une fonction qui, étant donnée un couple $\langle n, m\rangle$, retourne le couple $\langle m, m + 1\rangle$.
2. Donner un lambda-terme qui, étant donné un entier $n$, renvoie le couple $\langle n - 1, n\rangle$. *Indication : on remarquera qu'appliquer $n$ fois la fonction précédemment définie au couple $\langle 0, 0\rangle$ retourne le bon résultat.*
3. En déduire une fonction prec et une fonction minus, donnant l'opération $-$.

### 1.2.3 A propos de la récursivité

La récursivité est l'unique élément manquant pour montrer que le lambda-calcul est aussi expressif que les langages de programmation habituels. Une fonction récursive est une fonction de la forme $f(x) = g(f, x)$, c'est-à-dire que l'expression de $f$ s'exprime avec $f$.

**Exemple 1.5.** La fonction fact, factorielle, peut s'exprimer de la façon suivante :

$$\begin{array}{rccl} \text{fact} : & 0 & \longmapsto & 2 \\ & n & \longmapsto & n \times \text{fact}\ (n-1) \end{array}$$

Si nous voulions écrire cet exemple avec un lambda-terme, nous écririons

$$\text{fact} = \lambda n.\texttt{if eq0 } n \texttt{ then } \overline{1} \texttt{ else } n \times \text{fact}\ (\text{prec}\ n)$$

mais nous ne pouvons pas écrire ainsi fact dans la définition de fact. Il faut donc trouver un moyen d'exprimer cette équation d'une autre façon.

Pour cela, nous utiliserons un combinateur de point fixe. En effet, il suffit de trouver un point fixe à l'équation suivante :

$$\lambda g.\lambda n.\texttt{if eq0}\ n\ \texttt{then}\ \overline{1}\ \texttt{else}\ n \times g\ (\text{prec}\ n)$$

car en effet, un point fixe de cette fonction est une fonction $f$ telle que

$$f = \lambda n.\texttt{if eq0}\ n\ \texttt{then}\ \overline{1}\ \texttt{else}\ n \times f\ (\text{prec}\ n)$$

qui est la définition que nous voulons de fact.

**Définition 1.14** (Combinateur de point fixe)**.** Nous définissons le combinateur de point fixe $Y$ par le lambda-terme suivant :

$$Y = \lambda f.(\lambda x.f\ x\ x)\ (\lambda x.f\ x\ x)$$

Nous définissons le combinateur de point fixe $\Theta$ par le lambda-terme suivant :

$$\Theta = (\lambda f.\lambda x.(x\ (f\ f\ x)))\ (\lambda f.\lambda x.(x\ (f\ f\ x)))$$

**Exercice 1.12.** Montrer que pour tout lambda-terme $f$, $Y\ f =_\beta f(Y\ f)$ et que $\Theta f \to_\beta^* f(\Theta\ f)$.

Ainsi nous pouvons définir la fonction factorielle :

$$\text{fact} = Y\Big(\lambda g.\lambda n.\texttt{if eq0}\ n\ \texttt{then}\ \overline{1}\ \texttt{else}\ n \times g\ (\text{prec}\ n)\Big)$$

**Exercice 1.13.** Réduire fact 5.

**Exercice 1.14.** Cet exercice vise à coder en lambda-calcul la suite de Fibonacci.

1. Montrer que la relation
$$u_{n+2} = u_{n+1} + u_n$$
peut s'exprimer comme une relation de récurrence d'ordre 1 sur une autre suite.
*Indication : on considérera la suite des termes de la forme* $\langle u_i, u_{i+1}\rangle$.
2. En déduire un codage de la suite de Fibonacci définie par :
$$\begin{array}{rcl} \text{fib}\ \overline{0} & = & \overline{0} \\ \text{fib}\ \overline{1} & = & \overline{1} \\ \text{fib}\ (S\ S\ \overline{n}) & = & \text{fib}\ (S\ n) + \text{fib}\ n \end{array}$$

# 2 Le lambda-calcul simplement typé

Comme nous l'avons vu, le lambda-calcul possède un problème majeur : nous n'avons aucune assurance que calculer un terme convergera. Pour contrer ce fait, on utilise des types, qui sont des annotations de nos lambda-termes. Nous verrons dans cette partie ce qui constitue le lambda-calcul simplement typé, puis des résultats concernant ce lambda-calcul (principalement la forte normalisation). Enfin, nous explorerons des extensions classiques telles que l'ajout de types produits.

## 2.1 Définition des types

Nous travaillerons dans un premier temps dans un lambda-calcul ne possédant qu'un type de base, noté $\iota$. Un type précise dans quel univers est contenu un terme. Pour dire qu'un terme $M$ est annoté du type $\tau$, on note $M : \tau$.

**Définition 2.1** (Type)**.** On définit la grammaire des types :

$$\tau, \sigma ::= \iota \mid \tau \to \sigma$$

Nous lisons $\to$ avec une associativité à droite, ce qui signifie que $\tau_1 \to \tau_2 \to \tau_3 = \tau_1 \to (\tau_2 \to \tau_3)$.

De plus, nous noterons désormais $\lambda x^\tau.M$ plutôt que $\lambda x.M$ pour préciser le type de $x$.

Pour pouvoir définir la relation de typage, il convient de définir plusieurs éléments. Tout d'abord, intéressons-nous à ce qu'est un environnement de typage. C'est un ensemble (que l'on note comme une liste d'éléments séparés par des virgules) de couples $(x, \tau)$ où $x$ est une variable et $\tau$ un type. On notera $x : \tau$ les éléments de cet ensemble. Un environnement de typage pourra par exemple être $\Gamma = x : \iota, f : \iota \to \iota, g : \iota \to \iota \to \iota$. De plus, on note $\Gamma, x : \tau$ pour noter l'ensemble $\Gamma \cup \{(x, \tau)\}$. Cela nous permet de définir un jugement de typage, qui se fait dans un environnement.

**Définition 2.2** (Jugement de typage)**.** On définit $\Gamma \vdash M : \tau$ un jugement de typage par induction sur la structure de $M$ :

$$(x, \tau) \in \Gamma \ \frac{}{\Gamma \vdash x : \tau}\ \text{Ax} \qquad \frac{\Gamma, x : \tau \vdash M : \sigma}{\Gamma \vdash \lambda x^\tau.M : \tau \to \sigma}\ \text{Abs} \qquad \frac{\Gamma \vdash M : \sigma \to \tau \quad \Gamma \vdash N : \sigma}{\Gamma \vdash MN : \tau}\ \text{App}$$

Lorsque $\Gamma = \varnothing$, on note $\vdash M : \tau$ directement, et on dit alors que $M$ est typable, ou de type $\tau$.

**Exemple 2.1.**

- On montre que $\vdash \lambda x^\iota.x : \iota \to \iota$ :

$$\dfrac{\dfrac{}{x : \iota \vdash x : \iota}\ \text{Ax}}{\vdash \lambda x^\iota.x : \iota \to \iota}\ \text{Abs}$$

- On montre aussi que $\vdash \lambda f^{\iota \to \iota}.\lambda x^\iota.f\ x : (\iota \to \iota) \to \iota \to \iota$ :

$$\dfrac{\dfrac{\dfrac{\dfrac{}{f : \iota \to \iota, x : \iota \vdash f : \iota \to \iota}\ \text{Ax} \quad \dfrac{}{f : \iota \to \iota, x : \iota \vdash x : \iota}\ \text{Ax}}{f : \iota \to \iota, x : \iota \vdash f\ x : \iota}\ \text{App}}{f : \iota \to \iota \vdash .\lambda x^\iota.f\ x : (\iota \to \iota) \to \iota \to \iota}\ \text{Abs}}{\vdash \lambda f^{\iota \to \iota}.\lambda x^\iota.f\ x : (\iota \to \iota) \to \iota \to \iota}\ \text{Abs}$$

**Exercice 2.1.** Typer le lambda-terme $\lambda f^{\iota \to \iota}.\lambda g^{\iota \to \iota}.\lambda x^\iota.f\ (g\ x)$ puis généraliser ce typage à un lambda-terme identique pour lequel on remplace $\iota$ par un type indéterminé $\tau$.

## 2.2 Propriétés du lambda-calcul simplement typé

Tout d'abord, le lambda-calcul simplement typé, comme le non typé, est confluent et possède la propriété de Church-Rosser. Nous allons maintenant énumérer les autres propriétés de ce lambda-calcul.

**Théorème 2.1** (Préservation du typage)**.** *Soit un lambda-terme typé dans un contexte* $\Gamma \vdash M : \tau$ *et* $N$ *tel que* $M \to_\beta N$*. Alors* $\Gamma \vdash N : \tau$*.*

De plus, le lambda-calcul simplement typé est fortement normalisant.

**Définition 2.3.** Si $M$ est un terme bien typé, tel que $\vdash M : \tau$, alors toute suite de réduction depuis $M$ est finie (et converge donc vers une unique forme normale).

## 2.3 Extensions du langage

Le lambda-calcul simplement typé, dans son état actuel, possède un souci majeur : il ne permet pas d'exprimer l'arithmétique. Nous allons donc explorer une première extension du langage dans laquelle l'arithmétique est présente. Nous en profiterons pour ajouter la logique booléenne.

### 2.3.1 Ajout de l'arithmétique et des booléens

Cette partie traitera de l'ajout du type `int`et du type `bool`.Si nous avons codé les entiers naturels et les booléens dans le lambda-calcul non typé, le lambda-calcul typé, de par sa restriction forte sur la nature de ses objets, impose de définir les opérations arithmétiques et logiques de base dans la syntaxe du langage.

**Définition 2.4.** Nous définissons la grammaire des types et du langage :

$$\sigma, \tau ::= \texttt{int} \mid \texttt{bool} \mid \sigma \to \tau$$

$$M, N, P ::= x \mid \lambda x^\tau.M \mid MN \mid k \mid \top \mid \bot \mid M + N \mid M \times N \mid M - N \mid M \vee N \mid M \wedge N \mid \neg M \\ \mid M \leq N \mid \texttt{if } M \texttt{ then } N \texttt{ else } P$$

où $k \in \mathbb{N}$.

Nous devons alors redéfinir nos règles de typage en ajoutant celles nécessaires à l'ajout des opérations que nous avons.

**Définition 2.5** (Typage)**.** Les nouvelles règles de typages sont les suivantes :

$$\frac{}{\Gamma \vdash k : \texttt{int}} \qquad \frac{}{\Gamma \vdash \top : \texttt{bool}} \qquad \frac{}{\Gamma \vdash \bot : \texttt{bool}}$$

$$\frac{\Gamma \vdash M : \texttt{int} \quad \Gamma \vdash N : \texttt{int}}{\Gamma \vdash M + N : \texttt{int}} \qquad \frac{\Gamma \vdash M : \texttt{int} \quad \Gamma \vdash N : \texttt{int}}{\Gamma \vdash M \times N : \texttt{int}} \qquad \frac{\Gamma \vdash M : \texttt{int} \quad \Gamma \vdash N : \texttt{int}}{\Gamma \vdash M - N : \texttt{int}}$$

$$\frac{\Gamma \vdash M : \texttt{bool} \quad \Gamma \vdash N : \texttt{bool}}{\Gamma \vdash M \vee N : \texttt{bool}} \qquad \frac{\Gamma \vdash M : \texttt{bool} \quad \Gamma \vdash N : \texttt{bool}}{\Gamma \vdash M \wedge N : \texttt{bool}} \qquad \frac{\Gamma \vdash M : \texttt{bool}}{\Gamma \vdash \neg M : \texttt{bool}}$$

$$\frac{\Gamma \vdash M : \texttt{int} \quad \Gamma \vdash N : \texttt{int}}{\Gamma \vdash M \leq N : \texttt{bool}} \qquad \frac{\Gamma \vdash M : \texttt{bool} \quad \Gamma \vdash N : \tau \quad \Gamma \vdash P : \tau}{\Gamma \vdash \texttt{if } M \texttt{ then } N \texttt{ else } P : \tau}$$

Enfin, il convient aussi de redéfinir la réduction de termes. Nous allons noter $b$ un booléen et $k$ un entier, sous-entendu dans cette notation que les expressions associées sont réduites au maximum. Les ajouts à gauche d'une règles sont des conditions à vérifier pour appliquer la règle qui ne sont pas d'ordre syntaxique. Par exemple dans la première règle qui sera présentée, $k+k' = m$ sert simplement à préciser la valeur de $m$, mais cette condition n'est pas vérifiable par le système syntaxique. Évidemment, nous gardons pour autant la règle de la beta-réduction, mais nous allons y ajouter d'autres règles. Nous noterons simplement $\to$ la réduction ainsi faite.

**Définition 2.6** (Réduction)**.** Voici les règles de réduction supplémentaires :

$$\frac{M \to M'}{M + N \to M' + N} \qquad \frac{N \to N'}{M + N \to M + N'} \qquad k + k' = n \ \frac{}{k + k' \to n}$$

$$\frac{M \to M'}{M \times N \to M' \times N} \qquad \frac{N \to N'}{M \times N \to M \times N'} \qquad k \times k' = m \ \frac{}{k \times k' \to n}$$

$$\frac{M \to M'}{M - N \to M' - N} \qquad \frac{N \to N'}{M - N \to M - N'} \qquad k - k' = n \ \frac{}{k - k' \to n}$$

$$\frac{M \to M'}{M \vee N \to M' \vee N} \qquad \frac{N \to N'}{M \vee N \to M \vee N'} \qquad b \vee b' = c \ \frac{}{b \vee b' \to c}$$

$$\frac{M \to M'}{M \wedge N \to M' \wedge N} \qquad \frac{N \to N'}{M \wedge N \to M \wedge N'} \qquad b \wedge b' = c \ \frac{}{b \wedge b' \to c}$$

$$\frac{M \to M'}{\neg M \to \neg M'} \qquad \neg b = b' \ \frac{}{\neg b \to b'}$$

$$k \leq k' \ \frac{}{k \leq k' \to \top} \qquad k > k' \ \frac{}{k \leq k' \to \bot}$$

$$\frac{}{\texttt{if } \top \texttt{ then } M \texttt{ else } N \to M} \qquad \frac{}{\texttt{if } \bot \texttt{ then } M \texttt{ else } N \to N}$$

Les propriétés du lambda-calcul simplement typé sont encore conservées en utilisant ces ajouts.

**Exercice 2.2.**

- Effectuer la réduction du lambda-terme $\texttt{if } 0 = 27 \times (0 + 0) \texttt{ then } (\lambda x.x + 1)3 \texttt{ else } 5$
- Typer le lambda-terme $\lambda x.\texttt{if } (x - 5 \leq 0) \vee (6 \leq x) \texttt{ then } 1 \texttt{ else } 0$

### 2.3.2 Ajout des couples

Pour ajouter ce que nous avions utilisé dans le lambda-calcul non typé, il nous manque la récursion et les couples de valeurs. La récursion sera le problème d'une partie suivante, les couples seront étudiés ici.

Pour éviter de réécrire toute la grammaire que nous avions écrite précédemment, nous écrirons simplement $G$ pour l'ensemble des règles de construction définies.

**Définition 2.7.** La grammaire des types est la suivante :

$$\sigma, \tau ::= \texttt{int} \mid \texttt{bool} \mid \sigma \to \tau \mid \sigma \times \tau$$

Celles des termes vaut :

$$M, N, P ::= G \mid \langle M, N \rangle \mid \pi_1 M \mid \pi_2 M$$

**Définition 2.8** (Typage)**.** Les règles de typage ajoutées sont :

$$\frac{\Gamma \vdash M : \sigma \quad \Gamma \vdash N : \tau}{\Gamma \vdash \langle M, N \rangle : \sigma \times \tau} \qquad \frac{\Gamma \vdash M : \sigma \times \tau}{\Gamma \vdash \pi_1 M : \sigma} \qquad \frac{\Gamma \vdash M : \sigma \times \tau}{\Gamma \vdash \pi_2 M : \tau}$$

**Définition 2.9** (Réduction)**.** Les règles de réduction ajoutées sont :

$$\frac{M \to M'}{\langle M, N \rangle \to \langle M', N \rangle} \qquad \frac{N \to N'}{\langle M, N \rangle \to \langle M, N' \rangle}$$

$$\frac{}{\pi_1 \langle M, N \rangle \to M} \qquad \frac{}{\pi_2 \langle M, N \rangle \to N}$$

**Exercice 2.3.** Typer la fonction $\lambda x.\lambda y.\langle x, y \rangle$.

# 3 Le langage OCaml

Nous nous intéresserons maintenant à la syntaxe de OCaml et en donnerons une traduction en lambda-calcul simplement typé. Précisons ici que si OCaml fonctionne normalement avec du polymorphisme, nous nous restreindrons à en étudier une version sans polymorphisme.

Dans cette partie, nous détaillerons la syntaxe de OCaml (en donnant à ce moment-là les restrictions du langage comparé au vrai langage OCaml).

Nous allons désormais ajouter à notre langage la « fonction » $Y$. Nous pourrons donc noter $Y\ M$ un nouveau terme. La règle de typage est

$$\frac{\Gamma \vdash M : \tau \to \tau}{\Gamma \vdash Y\ M : \tau}$$

et la règle de réduction est

$$\frac{}{Y\ M \to M\ (Y\ M)}$$

Pour gagner en expressivité, nous devons alors sacrifier la sécurité de notre langage : en effet, il existe alors des lambda-termes qui ne sont pas normalisables.

**Exemple 3.1.** Le lambda-terme $M = Y\ (\lambda x^{\texttt{int}}.x+1)$ n'est pas normalisant. En effet, on peut montrer que $M \to^* M + n$ pour tout $n : \texttt{int}$. On construit ainsi une suite infinie de réductions.

**Exercice 3.1.** Trouver un lambda-terme typé $\Omega$ qui se réduise vers lui-même.

## 3.1 Syntaxe de OCaml

Nous allons maintenant préciser la syntaxe de notre version simplifiée de OCaml.

**Définition 3.1.** Un terme de OCaml est défini par la grammaire :

$$\begin{aligned} M, N, P ::= x \mid k \mid \top \mid \bot \mid \texttt{fun}\ x \to M \mid MN \mid \texttt{if}\ M\ \texttt{then}\ N\ \texttt{else}\ P \mid \texttt{let}\ x = M\ \texttt{in}\ N \\ \mid \texttt{let rec}\ x = M\ \texttt{in}\ N \mid M + N \mid M \times N \mid M - N \mid M \| N \mid M \texttt{\&\&} N \mid \texttt{not}\ M \mid M \leq N \mid \langle M, N \rangle \\ \mid \pi_1 M \mid \pi_2 M \mid \texttt{()} \end{aligned}$$

Les types dans OCaml sont :

$$\sigma, \tau ::= \texttt{int} \mid \texttt{bool} \mid \texttt{unit} \mid \sigma \times \tau \mid \sigma \to \tau$$

Le type `unit` correspond à un type à un élément, noté `()`.

Plutôt que de redéfinir de nouvelles règles de typage et de réduction, nous allons simplement traduire des termes de notre lambda-calcul simplement typé enrichi en des termes de notre OCaml.

**Définition 3.2** (Traductions en lambda-calcul)**.**

- On définit $\texttt{fun}\ x \to M$ comme $\lambda x.M$.
- On définit $\texttt{let}\ x = M\ \texttt{in}\ N$ comme $(\lambda x.N)\ M$.
- On définir $\texttt{let rec}\ x = M\ \texttt{in}\ N$ comme $(\lambda x.N)\ (Y\ (\lambda x.M))$.
- On définit $M \| N$ par $M \vee N$.
- On définit $M\texttt{\&\&}N$ comme $M \wedge N$.
- On définit $\texttt{not}M$ comme $\neg M$.

Enfin, nous définissons une syntaxe particulière pour alléger la lecture. L'expression $\texttt{let}\ f\ x = M\ \texttt{in}\ N$ signifie $\texttt{let}\ f = \texttt{fun}\ x \to M\ \texttt{in}\ N$ (et on peut définir ainsi en appliquant récursivement cette définition des fonctions à plusieurs variables : par exemple $\texttt{let}\ f\ x\ y = M\ \texttt{in}\ N$ sera traduit par $\texttt{let}\ f = \texttt{fun}\ x \to \texttt{fun}\ y \to M\ \texttt{in}\ N$).

**Exercice 3.2.** Déterminer les nouvelles règles de typage pour OCaml, à partir des règles de typage de leur expression équivalente en lambda-calcul simplement typé enrichi.

*Remarque.* La définition que nous avions faite en lambda-calcul non typé reste la même ici, pour la factorielle :

$$\texttt{let rec fact}\ n = \texttt{if}\ x = 0\ \texttt{then}\ 1\ \texttt{else}\ x \times \texttt{fact}\ (n-1)\ \texttt{in ()}$$

**Exercice 3.3.** Définir les règles de typage et de réduction de l'opérateur $=$, que nous ajouterons alors au langage.

## Deuxième partie

# Théorie des domaines

La théorie des domaines est une théorie visant à étudier certains types d'ensembles permettant de modéliser le lambda-calcul. Nous étudierons ici les domaines de Scott à partir des ordres partiels complets, puis nous étudierons la topologie liée à ces domaines.

## 4 Ordre et domaine

Nous supposons ici connues les notions basiques de théorie des ordres (relation d'ordre, borne supérieure, majorant, etc).

### 4.1 Définitions de base

**Définition 4.1** (Ordre partiel, dirigé)**.** Soit $(D, \leq)$ un ensemble ordonné. On dit qu'une partie $\Delta \subseteq D$ est filtrante si

$$\forall x, y \in \Delta, \exists z \in \Delta, x \leq z \wedge y \leq z$$

Un exemple classique de partie filtrante est celui des chaînes, que l'on peut voir comme des suites croissantes d'éléments. On notera $\Delta \subseteq_{\text{dir}} D$ pour dire que $\Delta$ est une partie filtrante de $D$.

On dit que $D$ est un ordre partiel dirigé complet si toute partie filtrante $\Delta \subseteq D$ admet une borne supérieure, qu'on notera $\bigvee \Delta$. Si de plus, $D$ possède un minorant, que l'on notera $\bot$, alors on dit que $D$ est un ordre partiel complet. On abrégera désormais « ordre partiel dirigé complet » en dcpo (*directed complete partial order*) et « ordre partiel complet » en cpo (*complete partial order*).

Définissons maintenant la notion adaptée de morphisme dans ce contexte.

**Définition 4.2** (Fonction croissante, continue)**.** Soient deux dcpo $D$ et $D'$ et $f : D \to D'$. On dit que $f$ est croissante lorsque

$$\forall x, y \in D, x \leq y \implies f(x) \leq f(y)$$

et continue lorsque $f$ est croissante et que

$$\forall \Delta \subseteq_{\text{dir}} D, f(\bigvee \Delta) = \bigvee f(\Delta)$$

**Exercice 4.1.** Pour s'assurer que la définition de fonction continue est valide, vérifier que l'image d'une partie filtrante par une fonction croissante est une partie filtrante de l'ensemble d'arrivée.

**Exercice 4.2.** Montrer que si $D$ est un dcpo, alors l'identité est continue sur $D$. Montrer que la composée de deux fonctions continues est continue.

De l'exercice précédent, on déduit que les dcpo forment une catégorie, notée **Dcpo** (cf. Définition 5.1). De plus, **Cpo** est la sous-catégorie pleine dont les objets sont les cpo.

Donnons des exemples classiques et utiles de cpo. La vérification que ceux-ci sont des cpo est laissée en exercice.

**Exemple 4.1.**

- Si $X$ est un ensemble, alors l'ensemble $X_\bot = X \cup \{\bot\}$ (où $\bot \notin X$) est un cpo en le munissant de l'ordre défini par

  $$x \leq y \iff x = \bot \vee x = y$$

  On appelle cela le domaine plat de $X$. Par exemple, le domaine $\mathbb{N}_\bot$ peut être représenté par la figure 1, indiquant l'ordre dans l'ensemble par une flèche.
- Si $X$ et $Y$ sont deux ensembles, alors l'ensemble des fonction partielles de $X$ dans $Y$, noté $X \rightharpoonup Y$, forme un cpo. L'ordre est donné par l'inclusion des graphes de fonction, c'est-à-dire que $f \leq g$ si $g$ est égale à $f$ là où $f$ est définie. Le minorant de cet ensemble est la fonction définie nulle part.

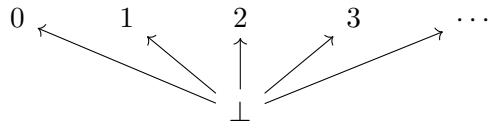


Figure 1 – Domaine $\mathbb{N}_\perp$

Nous pouvons déjà motiver notre étude des cpo par un premier résultat : une fonction continue d'un cpo dans lui-même possède un point fixe (ce qui fait des cpo de bons candidats pour modéliser le combinateur $Y$).

**Proposition 4.1.1.** *Soit $(D, \leq)$ un cpo et $f : D \to D$ continue. Alors $\bigvee_{n\in\mathbb{N}} f^n(\perp)$ est le plus petit pré-point fixe de $f$ (c'est-à-dire point $x$ tel que $f(x) \leq x$).*

*Démonstration.* Tout d'abord, cet élément existe car $(f^n(\perp))$ est une suite croissante d'éléments. En effet, comme $\perp \leq f(\perp)$, en appliquant $n$ fois $f$ qui est croissante, on en déduit que $f^n(\perp) \leq f^{n+1}(\perp)$. C'est un point fixe car

$$f\left(\bigvee_{n\in\mathbb{N}} f^n(\perp)\right) = \bigvee_{n\in\mathbb{N}} f^{n+1}(\perp) = \bigvee_{n\in\mathbb{N}} f^n(\perp)$$

De plus, si $f(x) \leq x$, alors on peut montrer par récurrence que $f^n(\perp) \leq x$. Le car $n = 0$ est évident par minoration de $\perp$, et si $f^n(\perp) \leq x$ alors $f^{n+1}(\perp) \leq f(x)$ donc $f^{n+1}(\perp) \leq x$ par transitivité. On en déduit par inégalité sur les bornes supérieures le résultat. □

**Exercice 4.3.** Soit $D$ un treillis complet, c'est-à-dire que $D$ est un ensemble ordonné tel que toute partie de $D$ a une borne supérieure et une borne inférieure, et soit $f : D \to D$ croissante. Montrer qu'elle a un plus petit point fixe, qui est $\bigwedge\{x \mid f(x) \leq x\}$, et que l'ensemble des points fixes de $f$ forme un treillis complet.

**Exercice 4.4.** Soit $D$ un cpo, $f : D \to D$ croissante. On définit par induction transfinie $f^0 = \perp$, $f^{\lambda+1} = f(f^\lambda)$ et pour $\lambda$ ordinal limite, $f^\lambda = \bigvee_{\alpha<\lambda} f^\alpha$. Montrer qu'il existe un ordinal $\mu$ tel que $f^\mu$ est le plus petit pré-point fixe de $f$. *Indication : un cpo étant un ensemble, il est plus petit que la classe des ordinaux.*

## 4.2 Élements finis

Intéressons-nous désormais aux éléments comportant de l'information finie. En effet, nous allons vouloir écrire nos éléments comme des limites d'éléments finis, par exemple pour définir une fonction comme la limite d'un graphe se remplissant au fur et à mesure. Pour cela, nous avons besoin de la notion d'élément compact

**Définition 4.3** (Élement compact)**.** Soit $D$ un dcpo. On appelle élément compact (ou fini) de $D$ un élément $d \in D$ tel que

$$\forall \Delta \subseteq_{\text{dir}} D, \left[d \leq \bigvee \Delta \implies \exists \delta \in \Delta, d \leq \delta\right]$$

On note l'ensemble des éléments compacts de $D$ par $\mathcal{K}(D)$.

On souhaite alors déterminer des dcpo où les éléments sont des limites de suites de compacts, d'où la notion d'algébricité.

**Définition 4.4** (Algébricité)**.** On dit d'un dcpo $D$ qu'il est algébrique si pour tout $x \in D$, l'ensemble $\Delta = \{d \mid d \in \mathcal{K}(D), d \leq x\}$ est une partie filtrante telle que $\bigvee \Delta = x$. On appelle les éléments de $\Delta$ les approximations de $x$, et on dit que $\mathcal{K}(D)$ est la base de $D$.

La sous-catégorie pleine de **Dcpo** contenant les dcpo algébriques est notée **Adcpo**, et de même **Acpo** est la sous-catégorie pleine de **Cpo** contenant les cpo algébriques.

**Exemple 4.2.** Nous avons vu que $X \rightharpoonup Y$ est un cpo, c'est en fait un acpo où les éléments compacts sont les fonctions de domaine de définition fini.

**Exercice 4.5.** Prouver que l'exemple précédent est bien un acpo.

Définissons maintenant une caractérisation de la continuité, qui sera plus naturelle en considérant les éléments compacts comme finis (en effet, on souhaite donc particulièrement regarder des limites dans $\mathcal{K}(D)$).

**Proposition 4.2.1** ($\epsilon\delta$-continuité)**.** *Soient $D$ et $D'$ deux adcpo. Alors*

- *Une fonction $f : D \to D'$ est continue si et seulement si elle est croissante et pour tout $d' \in \mathcal{K}(D')$ et $x \in D$ tel que $d' \leq f(x)$, il existe $d \in \mathcal{K}(D)$ tel que $d' \leq f(d)$.*
- *L'ensemble $\{(d, d') \in \mathcal{K}(D) \times \mathcal{K}(D') \mid d' \leq f(d)\}$ détermine le graphe de la fonction $f$.*

*Démonstration.* Prouvons le premier point.

Supposons $f$ continue. Soit $d'$ et $x$ pris comme dans l'énoncé, tels que $d' \leq f(x)$. Alors on trouve une partie filtrante $\Delta$ de $\mathcal{K}(D)$ dont $x$ est la borne supérieure, et comme $f(\Delta)$ est une partie filtrante de borne supérieure plus grande que $d'$, par compacité de $d'$ on trouve $f(d)$ tel que $d' \leq f(d)$, donc il existe bien un tel $d \in \mathcal{K}(D)$. Réciproquement, montrons que $f$ est continue. Soit $\Delta \subseteq_{\text{dir}} D$. Par croissance, on a $\bigvee f(\Delta) \leq f(\bigvee \Delta)$. Pour montrer l'inégalité inverse, il suffit de montrer que pour tout élément compact $d'$ tel que $d' \leq \bigvee f(\Delta)$ (par algébricité de $D'$), il existe $d \in \Delta$ tel que $d' \leq f(d)$. On construit d'abord $\delta \in \mathcal{K}(D)$ tel que $d' \leq f(\delta)$ et $\delta \leq \bigvee f(\Delta)$, et on construit alors $d$ à partir de la compacité de $\delta$.

On remarque que pour $x \in D$, on a

$$\begin{aligned} f(x) &= \bigvee \{d' \mid d' \in \mathcal{K}(D'), d' \leq f(x)\} \\ &= \bigvee \{d' \mid d' \in \mathcal{K}(D'), \exists d \in \mathcal{K}(D), d \leq x \wedge d' \leq f(d)\} \end{aligned}$$

D'où le résultat. □

## 4.3 Construire des dcpo

Nous avons déjà une intuition de ce que nous voulons pour interpréter notre langage OCaml : nous allons utiliser des cpo, et une fonction sera une fonction continue sur ces cpo. Pour pouvoir utiliser ce modèle, il nous faut cependant construire des ensembles adaptés aux ajouts que nous avons faits. Il faut donc pouvoir construire une notion qui permette d'avoir à la fois nos domaines et aussi des domaines pour le produit et l'espace des fonctions continues. Ces domaines sont les domaines de Scott, que nous verrons en fin de partie.

**Définition 4.5** (Produit de dcpo)**.** Soient $D$ et $D'$ deux dcpo. On définit leur produit $D \times D'$ en donnant l'ordre

$$(d, d') \leq (e, e') \iff d \leq e \wedge d' \leq e'$$

Si $D$ et $D'$ sont des cpo, alors $D \times D'$ est aussi un cpo.

*Démonstration.* Soit $\Delta \subseteq_{\text{dir}} D \times D'$. On définit $\Delta_D = \{d \mid \exists d' \in D', (d, d') \in \Delta\}$ et de même $\Delta_{D'}$. On peut alors définir $(\delta, \delta') = (\bigvee \Delta_D, \bigvee \Delta_{D'})$. On vérifie directement que $(\delta, \delta') = \bigvee \Delta$. Enfin, le minorant de deux cpo est $(\bot, \bot)$. □

Donnons un résultat primordial pour justifier la continuité des fonctions que nous utiliserons. En effet, si la continuité n'est en général pas simplement la continuité pour chaque argument, il se trouve que c'est le cas ici.

**Proposition 4.3.1.** *Soient $D, D', E$ des dcpo, et $f : D \times D' \to E$. Alors $f$ est continue si et seulement si les fonctions $f_x : y \mapsto f(x, y)$ et les fonctions $f_y : x \mapsto f(x, y)$ sont continues pour tout $x$ et tout $y$.*

*Démonstration.* Le sens direct se fait de façon en remarquant que si $\Delta \subseteq_{\text{dir}} D'$ alors pour tout $x \in D$, $(x, \Delta) = \{(x, \delta) \mid \delta \in \Delta\}$ est une partie filtrante de $D \times D'$. Alors

$$\bigvee f_x(\Delta) = \bigvee f(x, \Delta) = f \bigvee (x, \Delta) = f_x(\bigvee \Delta)$$

Réciproquement, soit $f$ continue en chaque argument. Alors soit $\Delta \subseteq_{\text{dir}} D \times D'$. On réutilise la définition précédente de $\Delta_D$ et $\Delta_{D'}$. Alors :

$$\begin{aligned} f(\bigvee \Delta) &= f(\bigvee \Delta_D, \bigvee \Delta_{D'}) = \bigvee f(\Delta_D, \bigvee \Delta_{D'}) \\ &= \bigvee \{\delta, \bigvee \Delta \mid \delta \in \Delta_D\} = \bigvee f(\Delta_D, \Delta_{D'}) \end{aligned}$$

Il convient alors de prouver que $\bigvee f(\Delta_D, \Delta_{D'}) = \bigvee f(\Delta)$. $\Delta \subseteq \Delta_D \times \Delta_{D'}$ d'où l'une des inégalités. Réciproquement, comme $\Delta$ est filtrant, pour chaque élément de $\Delta_D \times \Delta_{D'}$ il existe un majorant dans $\Delta$. □

Pour le produit, l'algébricité se passe correctement.

**Proposition 4.3.2.** *Soient $D, D'$ deux adcpo, alors $D \times D'$ est un adcpo et $\mathcal{K}(D \times D') = \mathcal{K}(D) \times \mathcal{K}(D')$.*

*Démonstration.* Prouvons d'abord l'égalité sur les éléments compacts.

Soit $(d, d') \in \mathcal{K}(D \times D')$. Soit $\Delta \subseteq_{\text{dir}} D$ tel que $d' \leq \bigvee \Delta$. Alors $(\Delta, d') \subseteq_{\text{dir}} D \times D'$ et $(d, d') \leq \bigvee(\Delta, d')$ donc par algébricité on trouve un élément $\delta$ de $\Delta$ tel que $d \leq \delta$, donc $d$ est algébrique. De même, on prouve que $d'$ est algébrique. Donc $\mathcal{K}(D \times D') \subseteq \mathcal{K}(D) \times \mathcal{K}(D')$.

Soit $(d, d') \in \mathcal{K}(D) \times \mathcal{K}(D')$. Soit $\Delta \subseteq_{\text{dir}} D \times D'$ tel que $(d, d') \leq \bigvee \Delta$. On sait que $\bigvee \Delta = \bigvee(\Delta_D, \Delta_{D'})$, donc on trouve par algébricité de $d$ et de $d'$, respectivement $\delta \in \Delta_D$ et $\delta' \in \Delta_{D'}$ tels que $d \leq \delta$ et $d' \leq \delta'$. Or par définition de $\Delta_D$ et $\Delta_{D'}$, on trouve $\alpha$ et $\alpha'$ tels que $(\delta, \alpha) \in \Delta$ et $(\alpha', \delta') \in \Delta$. Puisque $\Delta$ est filtrante, on trouve $(\beta, \beta')$ supérieur à ces deux valeur, dans $\Delta$. Donc on a trouvé $(d, d') \leq (\beta, \beta') \in \Delta$ : $(d, d')$ est algébrique dans $D \times D'$.

Montrons alors que $D \times D'$ est algébrique. Soit $(x, y) \in D \times D'$. La partie de $\mathcal{K}(D \times D')$ donnée par $\{(d, d') \mid (d, d') \in \mathcal{K}(D \times D'), (d, d') \leq (x, y)\}$ est filtrante car filtrante sur chaque coordonnée. De plus, En étudiant coordonnée par coordonnée, on en déduit bien que $(x, y)$ est la borne supérieure de cet ensemble. □

Intéressons-nous maintenant à l'espace fonctionnel d'un dcpo.

**Définition 4.6** (Espace des fonctions d'un dcpo)**.** L'ensemble $D \to D'$, constitué des fonctions continues de $D$ dans $D'$, pour $D$ et $D'$ deux dcpo, est aussi un dcpo, avec l'ordre suivant :

$$f \leq f' \iff \forall x \in D, f(x) \leq f'(x)$$

De plus, si $D'$ est un cpo, alors $D \to D'$ est un cpo.

*Démonstration.* Soit $\Delta \subseteq_{\text{dir}} D \to D'$. On pose $f : x \mapsto \bigvee \Delta(x)$. Montrons que cette fonction est bien continue. Soit $\Delta' \subseteq_{\text{dir}} D$. Alors

$$\begin{aligned} f(\bigvee \Delta') &= \bigvee \Delta(\bigvee \Delta') = \bigvee \{\bigvee g(\Delta') \mid g \in \Delta\} = \bigvee \Delta(\Delta') \\ &= \bigvee \{\bigvee \Delta(\delta) \mid \delta \in \Delta'\} = \bigvee f(\Delta') \end{aligned}$$

De plus, $x \mapsto \bot$ est un minorant de $D \to D'$ s'il existe. □

Les exercices suivants serviront pour l'interprétation catégorique du lambda-calcul.

**Exercice 4.6.** Soit $D$ un dcpo. Montrer que la fonction $\begin{array}{rccc} Y : & (D \to D) & \longrightarrow & D \\ & f & \longmapsto & \bigvee\limits_{n \in \mathbb{N}} f^n(\bot) \end{array}$ est continue.

**Exercice 4.7.** Soient $D$ et $D'$ deux dcpo. Montrer que les fonctions $\pi_1 : D \times D' \to D$ et $\pi_2 : D \times D' \to D'$, les deux projections, sont continues. Montrer aussi que la fonction $\langle -, - \rangle : D \to D' \to D \times D'$, $x \mapsto (y \mapsto \langle x, y \rangle)$ est continue.

**Exercice 4.8.** Soient $D$, $D'$ et $E$ trois dcpo. Montrer que la fonction $\mathrm{ev} : (D \to D') \times D \to D'$ donnée par $\mathrm{ev}(f, x) = f(x)$ est continue. Soit la fonction $\Lambda : ((D \times D') \to E) \to (D \to D' \to E)$ qui à $f$ associe $\Lambda(f) : x \mapsto (y \mapsto f(x, y))$, montrer que $\Lambda$ est continue.

Un souci arrive alors : si $D$ et $D'$ sont algébriques, $D \to D'$ ne l'est pas forcément. Pour autant, on peut dire plusieurs choses.

**Proposition 4.3.3** (Fonction en escalier)**.** *Soient $D$ et $D'$ deux cpo, et $(d, d') \in \mathcal{K}(D) \times \mathcal{K}(D')$ alors :*

- *La fonction $d \to d'$ est compact, avec la définition suivante :*

$$(d \to d')(x) = \begin{cases} d' & \text{si } d \leq x \\ \bot & \text{sinon} \end{cases}$$

- *Si $D$ et $D'$ sont algébrique, alors*

$$f = \bigvee \{ d \to d' \mid (d \to d') \leq f \}$$

*Démonstration.* Remarquons d'abord que la compacité de $d$ permet de déduire que $d \to d'$ est continue. De plus, pour tout $f : D \to D'$, $d \to d' \leq f$ si et seulement si $d' \leq f(d)$.

- Si $d \to d' \leq \bigvee \Delta$, alors $d' = (d \to d')(d) \leq \bigvee \{ f(d) \mid d \in \Delta \}$ d'où la conclusion par compacté de $d'$.
- On remarque que $\{ d \to d' \mid (d \to d') \leq f \} \leq g$ si et seulement si pour tout $d, d'$, $(d' \leq f(d) \implies d' \leq g(d))$ et si et seulement si $f \leq g$.

□

Le souci maintenant est que cet ensemble n'est pas filtrant. En effet, nous voulons donc pouvoir définir la borne supérieure d'un nombre fini de ces fonctions en escalier. Pour cela, nous ajoutons une condition.

**Définition 4.7** (Domaine de Scott)**.** On dit qu'un dcpo est borné complet si pour toute paire $(x, y)$ telle qu'il existe un majorant de $\{x, y\}$, il existe une borne supérieure à $\{x, y\}$.

On appelle un domaine de Scott un cpo algébrique borné complet et on note **S** la sous-catégorie pleine de **Acpo** contenant comme objet les domaines de Scott.

Nous admettrons alors un dernier résultat.

**Théorème 4.1.** *Si $D$ et $D'$ sont des domaines de Scott, alors $D \to D'$ est aussi un domaine de Scott, et les éléments algébriques de ce domaines sont exactement de la forme $(d_1 \to d'_1) \wedge (d_2 \to d'_2) \wedge \ldots \wedge (d_n \to d'_n)$ pour un $n$ fini.*

# 5 Interprétation dans une catégorie cartésienne fermée du lambda-calcul

Cette section réutilisera les résultats de la section précédente pour développer un cadre général d'interprétation catégorique. Nous allons donc faire correspondre à nos lambda-termes des fonctions dans une catégorie. Le choix d'une catégorie cartésienne fermée est naturel puisque c'est la notion la plus simple de catégorie possédant des exponentiations, c'est-à-dire des objets de la forme $a \to b$. Nous verrons ainsi, tout d'abord, la définition d'une catégorie cartésienne fermée, pour pouvoir ensuite donner l'interprétation d'un lambda-terme du lambda-calcul simplement typé (sans extension car celles-ci seront traitées dans la partie sur l'interprétation de OCaml).

## 5.1 Définitions

Rappelons la définition d'une catégorie.

**Définition 5.1** (Catégorie)**.** Une catégorie $\mathbf{C}$ est une classe composée :

- d'objets, dont on notera la classe $\mathbf{C}_0$.
- de flèches, dont on notera la classe $\mathbf{C}_1$. Chaque flèche $f$ possède un domaine, noté $\mathrm{dom}(f)$ et un codomaine, noté $\mathrm{codom}(f)$. On notera plus simplement $f : a \to b$ pour dire que $\mathrm{dom}(f) = a$ et $\mathrm{codom}(f) = b$.
- d'une opération de composition, associative, notée $\circ$, qui associe à deux flèches $f : a \to b$ et $g : b \to c$ une flèche $g \circ f : a \to c$.
- pour chaque objet $c$, d'une flèche $\mathrm{id}_c$ appelée identité de $c$ telle que pour toute flèche $f : a \to c, \mathrm{id}_c \circ f = f$ et pour toute flèche $g : c \to b, g \circ \mathrm{id}_c = g$.

Nous donnerons des exemples de catégories classiques et utiles dans notre étude.

**Exemple 5.1.**

- La catégorie **Set** des ensembles avec comme flèches les applications entre les ensembles.
- Les catégories **Dcpo** et **Cpo** avec comme flèches les application continues.
- Les catégories **Adcpo** et **Acpo** avec les mêmes flèches.
- La catégorie **S** des domaines de Scott avec toujours les applications continues.

*Remarque.* Le fait que les catégories précédentes en sont bien a été vérifié au long des exercices de la partie précédente, lorsqu'on justifiait la continuité des différentes fonctions.

Nous avons, de plus, besoin de trois éléments pour définir une catégorie cartésienne fermée : un objet terminal, un produit et une exponentiation. Nous allons donc définir ces termes dans un premier temps.

**Définition 5.2** (Objet terminal)**.** On appelle objet terminal d'une catégorie $\mathbf{C}$ un objet, noté 1, tel que pour tout objet $a \in \mathbf{C}_0$, il existe une unique flèche $!_a : a \to 1$.

**Définition 5.3** (Produit)**.** Soit une catégorie $\mathbf{C}$, deux objets $a$ et $b$. On appelle produit de $a$ et $b$, et on note $a \times b$, l'unique objet à isomorphisme près tel que pour tout objet $x$ et toute paire de flèche $f : x \to a, g : x \to b$ il existe une unique fonction, notée $\langle f, g \rangle$ qui fasse commuter le diagramme de la figure 2.

**Définition 5.4** (Exponentiation)**.** Soient $a$ et $b$ deux objets de $\mathbf{C}$. On appelle exponentielle de $a$ par $b$ l'objet $a^b$, représentant les fonctions $b \to a$. On se munit d'une fonction $\mathrm{ev} : a^b \times b \to a$ et on a la propriété universelle que pour toute fonction $f : c \times b \to a$, alors il existe une unique fonction $\Lambda(f)$ (appelée curryfication de $f$) telle que le diagramme de la figure 3 commute.

Une catégorie cartésienne fermée est donc une catégorie dans laquelle on a un élément terminale, tous les produits binaires (et donc tous les produits finis, par récurrence évidente) et toutes les exponentiations. Notre cadre d'étude, que sont les domaines de Scott, est une catégorie cartésienne fermée **S**, grâce aux exercices précédents.

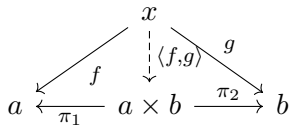


Figure 2 – Diagramme commutatif du produit

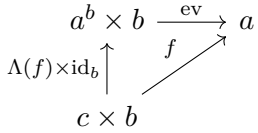


Figure 3 – Diagramme commutatif de l'exponentielle

## 5.2 Interprétation catégorique

Nous pouvons désormais définir l'interprétation catégorique du lambda-calcul simplement typé dans une CCC. Pour cela, on se fixe tout d'abord les objets de notre catégorie. Ceux-ci seront des interprétations de nos types. On notera $[\![-]\!]$ la fonction d'interprétation, associant à un objet syntaxique (type, terme...) une interprétation sémantique dans notre catégorique. Nous supposons donc que pour chaque type $\tau$ il existe un objet $[\![\tau]\!]$ associé dans lequel sera interprété notre type. Les morphismes seront les interprétations des jugements de typage.

Nous allons d'abord définir l'interprétation d'un contexte. Soit un contexte $\Gamma$ donné sous la forme $\Gamma = x_1 : \tau_1, \ldots, x_n : \tau_n$, on définit $[\![\Gamma]\!] = [\![\tau_1]\!] \times \ldots \times [\![\tau_n]\!]$. Si $\Gamma = \varnothing$, alors $[\![\Gamma]\!] = 1$. En effet, une flèche $1 \to a$ est exactement un élément de $a$, donc un jugement de la forme $\vdash M : \tau$ sera exactement un élément de $[\![\tau]\!]$. Nous pouvons maintenant, par induction sur la structure d'un lambda-terme simplement typé, définir l'interprétation d'un lambda-terme.

**Définition 5.5** (Interprétation)**.** On définit par induction l'interprétation d'un lambda-terme :

- Si $\Gamma \vdash x : \tau$, on note $i$ l'indice d'occurrence de $x$ dans $\Gamma$, alors

$$[\![\Gamma \vdash x : \tau]\!] = \pi_i$$

  l'interprétation d'une variable est donc une projection du contexte.

- Si $\Gamma \vdash \lambda x.M : \sigma \to \tau$, on note $[\![\Gamma, x : \sigma]\!] = [\![\Gamma]\!] \times [\![\sigma]\!]$, ce qui nous permet e curryfier notre fonction :

$$[\![\Gamma \vdash \lambda x.M : \sigma \to \tau]\!] = \Lambda[\![\Gamma, x : \sigma \vdash M : \tau]\!]$$

- Si $\Gamma \vdash M\ N : \tau$, nous avons juste à appliquer l'évaluation à $M$ et $N$ :

$$[\![\Gamma \vdash M\ N : \tau]\!] = \mathrm{ev} \circ \langle [\![\Gamma \vdash M : \sigma \to \tau]\!], [\![\Gamma \vdash N : \sigma]\!] \rangle$$

Ainsi, une catégorie cartésienne close, et en particulier **S**, permet d'interpréter les éléments de base du lambda-calcul simplement typé.

## Troisième partie
# Conclusion

Nous allons donc pouvoir construire une interprétation du langage OCaml dans la catégorie **S** des domaines de Scott. Plus précisément, nous allons construire la sous-catégorie **OCaml** dont les objets sont :

$$\sigma, \tau ::= \mathbb{N}_\perp \mid \mathbb{B}_\perp \mid \mathbb{U}_\perp \mid \sigma \to \tau \mid \sigma \times \tau$$

et dont les morphismes sont les interprétations dans la CCC ainsi construite des termes définis par la syntaxe de **OCaml**. Nous devons donc définir une fonction d'interprétation $[\![-]\!]$ qui à un terme syntaxique associe le morphisme de **OCaml** qui lui correspond. On confondra ici un objet de la forme $1 \to A$ et un élément du domaine de Scott $A$. Par la structure de CCC, nous avons déjà l'interprétation de $\texttt{fun}\ x \to M$, de $x$ et de $M\ N$, de $\langle M, N\rangle$ et des projections. $\texttt{fun}\ x \to M$ nous permet de définir $\texttt{let}\ \ x = e\ \ \texttt{in}\ \ e'$. Nous allons détailler l'interprétation des différentes autres constructions.

L'interprétation des constantes booléennes est celle des élément de $\mathbb{B}$ dans $\mathbb{B}_\perp$. De même, $k \in \mathbb{N}$ est interprété par lui-même dans $\mathbb{N}_\perp$ et `()` est interprété par le seul élément de $\mathbb{U}$ dans $\mathbb{U}_\perp$. On notera désormais $\mathbb{B}_\perp = \{\texttt{true}, \texttt{false}, \perp\}$ pour ne pas confondre `false` et $\perp$.

**Proposition 5.2.1.** *On appelle fonction stricte une fonction $f : D \to D$ telle que $f(\perp) = \perp$. Une fonction stricte sur un domaine plat est continue.*

*Démonstration.* La croissance est respectée puisque si $x \leq y$ alors soit $x = \perp$ auquel cas $f(x) \leq f(y)$ soit $x = y$ auquel cas $f(x) \leq f(y)$. Une partie filtrante $\Delta$ contiendra au plus un élément différent de $\perp$. Si elle ne contient que $\perp$, alors l'image sera $\perp$ et la continuité est bien respectée, et s'il existe $a \neq \perp$, alors $\bigvee \Delta = a$ donc $\bigvee f(\Delta) = f(a)$. □

On en déduit donc que l'interprétation de not par la négation booléenne stricte est continue, on peut donc définir $[\![\text{not}]\!] : \mathbb{B}_\perp \to \mathbb{B}_\perp$.

Nous allons maintenant définir les opérations binaires. Celles-ci seront définie uniquement lorsque leurs deux arguments sont définis. Par exemple, + sera défini comme $\perp$ si l'un des arguments est $\perp$ et comme la somme de ses arguments sinon.

Ainsi on définit $+, -, \times, \texttt{\&\&}, \|, \leq$ comme des fonctions continues car elles sont continues en chaque argument (en fixant un argument, on a une fonction stricte sur un domaine plat).

L'interprétation du $\texttt{let rec}\ \ x = e\ \ \texttt{in}\ \ e'$ se fait à partir de celle de $Y$, qui est le combinateur de point fixe $Y(f) = \bigvee_{n \in \mathbb{N}} f^n(\perp)$.

Il nous reste enfin à définir l'interprétation $[\![\texttt{if}\ \ \texttt{then}\ \ \texttt{else}\ ]\!]$. Cette fonction est de la forme $\mathbb{B}_\perp \times D \times D \to D$ où $D$ est un domaine de Scott et renvoie $\perp$ si l'un des arguments est $\perp$ et fait sinon l'association suivante : $(\texttt{true}, e, e') \mapsto e$ et $(\texttt{false}, e, e') \mapsto e'$. Il suffit alors d'utiliser la continuité en chaque argument. En effet, pour $\mathbb{B}_\perp$ on a une fonction stricte sur un domaine plat et pour chaque $D$ la fonction est une projection, qui est donc continue.

Nous pouvons donc donner une sémantique de nos lambda-termes, cf. figure 4. Nous écrirons $\gamma$ pour un élément d'un contexte donné, $\gamma \in [\![\Gamma]\!]$.

**Exemple 5.2.** Donnons la sémantique du terme

$$\texttt{let rec fact} = \texttt{fun}\ x \to \texttt{if}\ \ x = 0\ \ \texttt{then}\ \ 1\ \ \texttt{else}\ \ x \times \texttt{fact}\ (x-1)\ \ \texttt{in fact}$$

$$[\![\texttt{let rec fact} = \texttt{fun}\ x \to \texttt{if}\ \ x = 0\ \ \texttt{then}\ \ 1\ \ \texttt{else}\ \ x \times \texttt{fact}\ (x-1)\ \ \texttt{in fact}]\!] = \bigvee_{n \in \mathbb{N}} \Big[ g \mapsto x \mapsto [\![\texttt{if then else}]\!] \circ \langle [\![=]\!] \circ \langle 0, x\rangle, 1, [\![\times]\!] \circ \langle x, \text{ev} \circ \langle g, [\![-]\!] \circ \langle x, 1\rangle\rangle\rangle \Big]^n (\perp)$$

$$
\begin{aligned}
[\![\Gamma \vdash x : \tau]\!] &= \pi_i \\
[\![\Gamma \vdash k : \texttt{int}]\!] &= \gamma \mapsto k \\
[\![\Gamma \vdash \texttt{true} : \texttt{bool}]\!] &= \gamma \mapsto \texttt{true} \\
[\![\Gamma \vdash \texttt{false} : \texttt{bool}]\!] &= \gamma \mapsto \texttt{true} \\
[\![\Gamma \vdash \texttt{fun}\ x \to M : \sigma \to \tau]\!] &= \Lambda[\![\Gamma, x : \sigma \vdash M : \tau]\!] \\
[\![\Gamma \vdash M\ N : \tau]\!] &= \mathrm{ev} \circ \langle [\![\Gamma \vdash M : \sigma \to \tau]\!], [\![\Gamma \vdash N : \sigma]\!] \rangle \\
[\![\Gamma \vdash \texttt{if}\ M\ \texttt{then}\ N\ \texttt{else}\ P : \tau]\!] &= [\![\texttt{if then else}]\!] \circ \langle [\![\Gamma \vdash M : \texttt{bool}]\!], [\![\Gamma \vdash N : \tau]\!], [\![\Gamma \vdash P : \tau]\!] \rangle \\
[\![\Gamma \vdash \texttt{let}\ x = M\ \texttt{in}\ N : \tau]\!] &= \mathrm{ev} \circ \langle \Lambda[\![\Gamma, x : \sigma \vdash N : \tau]\!], [\![\Gamma \vdash M : \sigma]\!] \rangle \\
[\![\Gamma \vdash \texttt{let rec}\ x = M\ \texttt{in}\ N : \tau]\!] &= \mathrm{ev} \circ \langle \Lambda[\![\Gamma, x : \sigma \vdash N : \tau]\!], \bigvee_{n \in \mathbb{N}} \Lambda[\![\Gamma, x : \sigma \vdash M : \sigma \to \sigma]\!]^n(\bot_{[\![\sigma]\!]}) \rangle \\
[\![\Gamma \vdash M + N : \texttt{int}]\!] &= [\![+]\!] \circ \langle [\![\Gamma \vdash M : \texttt{int}]\!], [\![\Gamma \vdash N : \texttt{int}]\!] \rangle \\
[\![\Gamma \vdash M - N : \texttt{int}]\!] &= [\![-]\!] \circ \langle [\![\Gamma \vdash M : \texttt{int}]\!], [\![\Gamma \vdash N : \texttt{int}]\!] \rangle \\
[\![\Gamma \vdash M \times N : \texttt{int}]\!] &= [\![\times]\!] \circ \langle [\![\Gamma \vdash M : \texttt{int}]\!], [\![\Gamma \vdash N : \texttt{int}]\!] \rangle \\
[\![\Gamma \vdash M || N : \texttt{bool}]\!] &= [\![||]\!] \circ \langle [\![\Gamma \vdash M : \texttt{bool}]\!], [\![\Gamma \vdash N : \texttt{bool}]\!] \rangle \\
[\![\Gamma \vdash M \texttt{\&\&} N : \texttt{bool}]\!] &= [\![\texttt{\&\&}]\!] \circ \langle [\![\Gamma \vdash M : \texttt{bool}]\!], [\![\Gamma \vdash N : \texttt{bool}]\!] \rangle \\
[\![\Gamma \vdash \texttt{not}(M) : \texttt{bool}]\!] &= [\![\texttt{not}]\!] \circ [\![\Gamma \vdash M : \texttt{bool}]\!] \\
[\![\Gamma \vdash M \leq N : \texttt{bool}]\!] &= [\![\leq]\!] \circ \langle [\![\Gamma \vdash M : \texttt{int}]\!], [\![\Gamma \vdash N : \texttt{int}]\!] \rangle \\
[\![\Gamma \vdash \langle M, N \rangle : \sigma \times \tau]\!] &= \langle [\![\Gamma \vdash M : \sigma]\!], [\![\Gamma \vdash N : \tau]\!] \rangle \\
[\![\Gamma \vdash \pi_1 M : \tau]\!] &= \mathrm{ev} \circ \langle \pi_1, [\![\Gamma \vdash M : \tau \times \sigma]\!] \rangle \\
[\![\Gamma \vdash \pi_2 M : \tau]\!] &= \mathrm{ev} \circ \langle \pi_2, [\![\Gamma \vdash M : \tau \times \sigma]\!] \rangle
\end{aligned}
$$

Figure 4 – Sémantique de OCaml