ἡ ὕβρις

Continuum mathématique des ENS

Hwīlum cynniges þegn
guma gilphlæden, gidda genyndig,
sē ðe ealfela ealdgesegena
worn gemunde, word ōþer fand
sō ðe gebunden· secg eft ongan
sīð Bēowulfes snyttrum styrian
ond on spēd wrecan spel gerāde,
wordum wrixlan

Beowulf, v. 866–874

Éditorial

Trouver une victime pour écrire l'éditorial

Sommaire

Le comité de rédaction attend vos articles

Bases de Gröbner

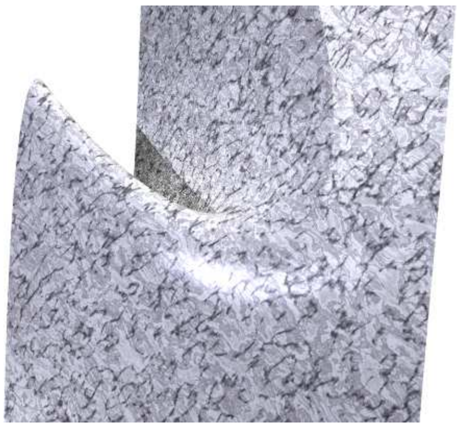
Vincent Zoonekynd
zoonek@math.jussieu.fr

Résumé

Nous exposons la notion de base de Gröbner, les principaux algorithmes qui la font intervenir (généralisations de la division euclidienne et du pivot de Gauss) et ses applications pour des calculs sur des polynômes à plusieurs indéterminées. Notre présentation s'inspire fortement de [CLO].

Introduction

Alors que tout un chacun manipule avec dextérité les polynômes à une indéterminée, nous restons particulièrement perplexes face à des polynômes à plusieurs indéterminées — tout au plus les polynômes symétriques nous sont-ils familiers. Ils sont pourtant omniprésents : par exemple, les mouvements du bras d'un robot sont décrits par des polynômes. La raison de la complexité du sujet est la suivante : l'étude des polynômes se ramène à l'étude de l'ensemble de leurs zéros. Pour les polynômes à une indéterminée, ce sont des ensembles finis, mais pour les polynômes à plusieurs indéterminées, c'est bien plus compliqué.



Les zéros du polynôme $y^2 - x^3 - zx$

Néanmoins, nous allons voir qu'une généralisation très simple de la division euclidienne permet d'effectuer des calculs dans l'anneau $k[x_1, \ldots, x_n]$ des polynômes à plusieurs indéterminées.

Division euclidienne

Commençons par rappeler la division euclidienne des polynômes à une indéterminée. Soient f et g de tels polynômes, dont on écrit les termes de plus haut degré en premier. On cherche à diviser f par g. Si le terme dominant de g ne divise pas celui de f, la division est finie : le quotient est nul et le reste est égal à f. En cas contraire, soit q le quotient de ces termes dominants ; on l'ajoute au quotient, on remplace f par $f - qg$ et on recommence.

Regardons sur un exemple : tentons de diviser $f = x^5$ par $g = x^2 + x + 1$. Le premier terme du quotient est $x^5/x^2 = x^3$ et le premier reste est

$$x^5 - x^3(x^2 + x + 1) = -x^4 - x^3,$$

ce qui s'écrit

$$f = x^3 g + (-x^4 - x^3).$$

On est maintenant amené à diviser $-x^4 - x^3$ par $x^2 + x + 1$, ce qui donne

$$-x^4 - x^3 = -x^2(x^2 + x + 1) + x.$$

Finalement, on cherche à diviser x par $x^2 + x + 1$, mais on ne peut pas : la division est finie et son résultat s'écrit

$$f = (x^3 - x^2)g + x.$$

On peut même poser la division.

$$\begin{array}{r|l}
x^5 \qquad\qquad & x^2 + x + 1 \\ \cline{2-2}
x^5 + x^4 + x^3 \qquad & x^3 - x^2 \\
- x^4 - x^3 \qquad & \\
- x^4 - x^3 - x & \\
x &
\end{array}$$

Avec plusieurs variables

Peut-on généraliser cet algorithme aux polynômes à plusieurs variables ? Le seul problème est de définir quel est le terme dominant d'un tel polynôme : il nous faut un ordre entre ses monômes — nous prendrons l'ordre lexicographique. Le même algorithme est maintenant utilisable.

On peut aussi vouloir diviser un polynôme f par plusieurs polynômes $g_1, \ldots, g_m$, *i.e.*, chercher à écrire

$$f = q_1 g_1 + \cdots + q_m g_m + r.$$

Il suffit de diviser par le permier polynôme et, quand on ne peut plus, de prendre le suivant et ainsi de suite. On s'arrête quand on ne peut plus diviser par aucun des polynômes.

Regardons sur un exemple. Cherchons à diviser $f = x^2y - xy$ par $g_1 = x^2 + y^2$ et $g_2 = y + 2$. Commençons par effectuer la division de f par g_1 ; il vient

$$f = yg_1 - xy - y^3.$$

Comme le reste n'est pas divisible par g_1, on essaye de le diviser par g_2, il vient alors

$$f = yg_1 - xg_2 + (2x - y^3).$$

Comme le reste a maintenant changé, regardons s'il ne serait pas à nouveau divisible par g_1 : ce n'est pas le cas. La division semble alors terminée, car on ne peut plus diviser ni par g_1 ni par g_2.

En fait, si on insiste, on peut quand même la poursuivre : le seul problème est que le terme dominant du reste, $2x$, n'est pas divisible par le terme dominant de g_1 ou g_2, *i.e.*, x^2 ou y. Mais si on met ce terme de côté, on peut continuer, en divisant $-y^3$ par g_1 et g_2 ; après calculs, il vient

$$f = yg_1 + (-x - y^2 + 2y - 4)g_2 + (2x + 8).$$

Cette fois-ci, on ne peut vraiment plus continuer, la division est terminée.

On peut remarquer que l'on aurait obtenu un resultat different en divisant par (g_2,g_1), en l'occurrence

$$f = (x^2 - x + 2y - 4)g_2 - 2g_1 + (2x + 8).$$

Neanmoins, le reste est le même : est-ce toujours le cas ? Eh bien, non ; si on divise $f = x^2y^2$ par $g_1 = xy$ et $g_2 = xy + y$, on n'obtient pas le même reste selon l'ordre des diviseurs.

$$f = xyg_1 + 0 \cdot g_2 + 0$$
$$f = (xy - y)g_2 + 0 \cdot g_1 + y^2.$$

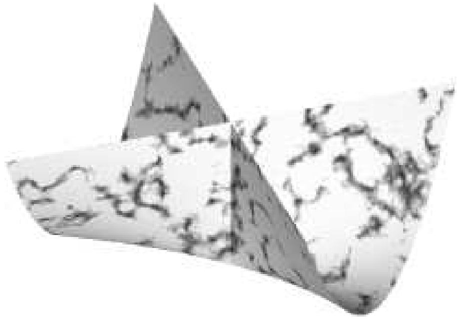
Interlude : les idéaux

L'un des buts — non encore atteint, patience — de l'algorithme précédent pouvait être de chercher si le polynôme f pouvait s'écrire

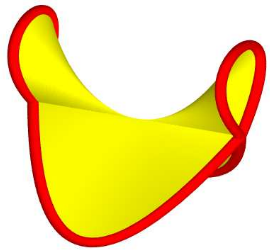
$$f = q_1g_1 + q_2g_2.$$

L'ensemble des polynômes de cette forme s'appelle l'*idéal* engendré par g_1 et g_2 et se note $\langle g_1,g_2\rangle$. Pour les polynômes à une indéterminée, nous nous étions limités à la division par un seul polynôme, car tous les idéaux de $k[x]$ sont de la forme $\langle g\rangle$ pour un polynôme g ; mais ce n'est plus vrai pour les polynômes à plusieurs indéterminées : c'est pour cette raison que nous avons tenté de diviser par un ensemble de polynômes.

Cette notion d'idéal a une interprétation géométrique : à un idéal $I \subset k[x_1,\dots,x_n]$ on associe l'ensemble des points de $\mathbf{C}^n$ qui annulent tous les éléments de I. Si on a un système de générateurs de I, il suffit en fait de demander les points qui annulent ces générateurs. Les parties de $\mathbf{C}^n$ ainsi obtenues sont appelées *variétés affines*. Les figures suivantes donnent quelques exemples (sur $\mathbf{R}$). On pourra remarquer que des idéaux différents peuvent définir la même partie de $\mathbf{C}^n$, c'est par exemple le cas de $\langle x\rangle$ et $\langle x^2\rangle$.



La surface d'équation $x^2 - y^2z = 0$



Une courbe définie par les équations $x^2 - y^2z^2 + z^3 = 0$ et $x^2 + y^2 + z^2 = 1$, i.e.*, à l'intersection de deux surfaces.*

Bases de Gröbner

La raison pour laquelle le reste de notre division n'est pas uniquement déterminé est la suivante : au cours de la division, le reste peut être une de combinaison « linéaire » des générateurs de notre idéal faisant disparaître les termes dominants et rendant ainsi la division impossible. Dans l'exemple précédent,

$$g_2 - g_1 = y$$

donne un reste non nul par la division par g_1 et g_2 : son terme dominant ne divise ni celui de g_1 ni celui de g_2. Pour pallier ce problème, nous allons agrandir la famille génératrice de l'idéal pour y inclure de telles combinaisons linéaires.

Plus formellement, étant donnés deux polynômes f et g, de termes dominants respectifs ax^α et bx^β, où α et β sont des multi-indices, on note $x^{\alpha\vee\beta}$ le plus petit multiple commun de x^α et x^β et on considère leur *polynôme* S, défini par

$$S(f,g) = bx^{\alpha\vee\beta-\alpha}f + ax^{\alpha\vee\beta-\beta}g.$$

C'est la combinaison « linéaire » la plus simple qui fait disparaître les termes dominants. On dit qu'un ensemble $G = \{f_1,\dots,f_n\}$ de polynômes est une *base de Gröbner* si la division du polynôme S de deux quelconques d'entre eux par G donne un reste nul.

Dès lors, pour obtenir une base de Gröbner, on procède ainsi : on part d'une famille de polynômes et on rajoute les restes des S-polynômes jusqu'à ce que tous ces restes soient nuls (on peut montrer que cet algorithme se termine).

Regardons ce qui se passe sur un exemple, l'idéal engendré par

$$\begin{aligned} f_1 &= x^3 - 2xy \\ f_2 &= x^2y - 2y^2 + x. \end{aligned}$$

Le S-polynôme de f_1 et f_2 est $-x^2$ et le reste de sa division par (f_1,f_2) est $-x^2$. On pose donc

$$f_3 = -x^2.$$

Le S-polynôme de f_1 et f_3 est $-2xy$ et le reste de sa division par (f_1,f_2,f_3) est $-2xy$; on pose donc

$$f_4 = -2xy.$$

Le S-polynôme de f_1 et f_4 est $-2xy^2$ mais le reste de sa division par (f_1,f_2,f_3,f_4) est nul. Par contre le S-polynôme de f_1 et f_4 est $x - 2y^2$ et le reste de sa division par (f_1,f_2,f_3,f_4) est $x - 2y^2$; on pose donc

$$f_5 = x - 2y^2.$$

Remarquons maintenant que le S-polynôme de f_1 et f_5 est $2x^2y^2 - 2xy$ et que son reste par la division par $(f_1,\dots,f_5)$ est $4y^3$; on pose donc

$$f_6 = 4y^3.$$

On peut désormais vérifier que les restes de tous les S-polynômes sont nuls : on a enfin obtenu une base de Gröbner.

On remarquera que l'on peut enlever certains de ces polynômes, en l'occurrence ne retenir que g_4 et g_5, tout en gardant une base de Gröbner — mais nous ne nous attarderons pas sur ce point.

Nous admettrons le résultat suivant : *Soit $g_1,\dots,g_n$ une base de Gröbner, un polynôme f est dans l'idéal $\langle g_1,\dots,g_n\rangle$ si et seulement si le reste de sa division par $g_1,\dots,g_n$ est nul.*

Applications

Nous donnons maintenant quelques applications des bases de Gröbner, sous forme d'exerices. Certains sont résolus, les autres sont laissés à la sagacité du lecteur. Nous utiliserons le logiciel de calcul formel Mupad, [M], qui est gratuit.

> *Le polynôme $x^2y - 2xy + 1$ est-il dans l'idéal engendré par $x^2 - x + 2y^2$ et $x + 2y - 1$?*

Il suffit de chercher une base de Gröbner de notre idéal et de calculer le reste de la division.

```
export(groebner):
p1:= poly(x^2 - x + 2*y^2, [x,y]):
p2:= poly(x + 2*y - 1, [x,y]):
G:= gbasis([p1,p2], LexOrder);
```

$$x + 2y - 1, \quad -y + 3y^2$$

```
p:= poly(x^2*y - 2*x*y + 1, [x,y]):
normalf(p, G, LexOrder);
```

$$5y/9 - 1$$

Comme le reste est non nul, on sait que le polynôme n'est pas dans l'idéal.

> *Comment l'ordinateur résout-il le système d'équations suivant ?*

$$\begin{aligned} x^2 - x + 2y^2 &= 0 \\ x^2 + y^2 &= 1 \end{aligned}$$

Remarquons tout d'abord que l'ordinateur sait résoudre ce système tout seul.

```
solve({x^2 - x + 2*y^2, x^2 + y^2-1});
```

$$(-2,i\sqrt{3}), \quad (-2,-i\sqrt{3}), \quad (1,0)$$

Mais comment fait-il ? Procède-t-il par tâtonnements et substitutions, comme on nous l'apprend à l'école ? Utilise-t-il une méthode plus systématique ? Comme vous l'avez dèjà deviné, il s'agit de bases de Gröbner. Résoudre ce système revient à chercher les points (x,y,z) qui annullent tous les polynômes de l'idéal $\langle g_1,g_2\rangle$ et il suffit de prendre une famille génératrice, par exemple une base de Gröbner pour l'ordre lexicographique.

```
p1:=poly(x^2 - x + 2*y^2,[x,y]):
p2:=poly(x^2 + y^2-1,[x,y]):
gbasis([p1,p2], LexOrder);
```

$$x - y^2 - 1, \quad 3y^2 + y^4$$

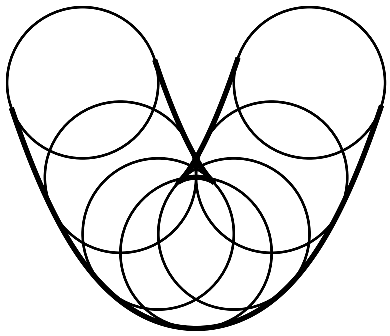
Le système que l'on obtient est bien plus agréable : la variable x a été éliminée de la seconde équation. Nous venons de voir que les bases de Gröbner permettent de trianguler des systèmes d'équations polynômiales (et non plus simplement linéaires, comme pour le pivot de Gauss), que l'on peut ensuite résoudre par substitutions.

Donner l'équation de l'enveloppe des cercles

$$(x-t)^2 + (y-t^2)^2 = 1.$$

Nous voulons éliminer la variable t dans le système suivant.

$$(x-t)^2 + (y-t^2)^2 = 1$$
$$(t-x) - 2t(y-t^2) = 0$$



Il suffit de calculer une base de Gröbner pour l'ordre lexicographique.

```
p1:=poly((x-t)^2+(y-t^2)^2-1,[t,x,y]):
p2:=poly(t-x -2*t*(y-t^2), [t,x,y]);
gbasis([p1,p2], LexOrder);
%1[5]
```

On obtient un résultat : l'équation implicite d'une courbe, que nous ne recopions pas. Mais attention : tout ce que nous savons, c'est que *si* un point est solution de nos équations, *alors* il est sur la courbe d'équation ci-dessus. La réciproque est fausse en général, mais nous éluderons le problème. Signalons simplement que les polynômes de la base de Gröbner trouvée ne faisant intervenir que les variables x et y engendrent l'idéal $I \cap k[x,y]$; en d'autres termes, ils décrivent la plus petite variété algébrique contenant l'enveloppe recherchée.

Quels sont les points singuliers de l'enveloppe que l'on vient de trouver ?

Trouver une équation implicite du tore.

Soient I et J des idéaux de $k[x_1,\dots,x_n]$; montrer que $I \cap J = \langle 1 - y_1 - y_2, Iy_1, Jy_2 \rangle \cap k[x_1,\dots,x_n]$. En déduire un algorithme permettant de calculer l'intersection de deux idéaux.

Pour en savoir plus...

[BW] B. Buchberger and F. Winkler, *Gröbner bases and applications*, LMSLNS 251, Cambridge University Press, 1998.

[CLO] D. Cox, J. Little, and D. O'Shea, *Ideals, varieties and algorithms*, Springer–Verlag, 1992.

[E] D. Eisenbud, *Commutative algebra with a view toward algebraic geometry*, Springer–Verlag, 1995.

[M] `http://www.mupad.de/`.

[S] B. Sturmfels, *Algorithms in invariant theory*, Springer–Verlag, 1993.

Lambda calcul

Alexandre Boisseau
boisseau@dptmaths.ens-cachan.fr

Le lambda-calcul a été développé en 1930 par Church dans un but double :

- d'une part décrire un formalisme de représentation des fonctions mathématiques,
- d'autre part lui adjoindre quelques principes de logique afin de développer une théorie qui puisse faire office de fondements aux mathématiques.

Actuellement, les objets que manipule le mathématicien ne sont que des ensembles. Par exemple un nombre réel est un ensemble de rationnels (précisément, l'ensemble des rationnels qui le minorent), un nombre rationnel est lui-même un couple de deux entiers, or le couple (a,b) est défini comme étant l'ensemble $\{a,\{a,b\}\}$. Enfin un entier est aussi un ensemble, 0 étant par définition l'ensemble vide et si $0,\ldots,n$ sont définis, on pose $n+1=\{0,\ldots,n\}$. Ces ensembles sont les objets de base et pour les manipuler le mathématicien dispose des ce qu'on appelle communément les «axiomes de la théorie des ensembles». L'alternative recherchée était donc de fonder les mathématiques non plus sur les ensembles mais sur les fonctions. On verra en effet plus loin comment on peut représenter les objets mathématiques usuels (couples, entiers, etc.) par des termes du lambda-calcul.

Kleene et Rosser ont montré un 1935 que le système était inconsistant et le second but n'a par conséquent pas pu être atteint. Par contre il s'est avéré que les fonctions obtenues de cette manière décrivaient exactement la classe des fonctions «effectivement calculables» (*i.e.*calculables par une machine de Turing). Ce que nous allons faire maintenant, c'est tout d'abord expliquer ce que sont les termes du lambda-calcul puis ensuite nous verrons comment les utiliser pour programmer de manière effective.

Construction des termes

On dispose à la base d'un ensemble X (infini mais dénombrable) dont les éléments sont appelés des variables. Ces variables sont des termes du lambda-calcul :

- $X \subset \lambda[X]$.

Si u et v sont deux termes, (uv) est un terme qui représente l'application du terme u au terme v :

- $u,v \in \lambda[X] \Rightarrow (uv) \in \lambda[X]$.

Si u est un terme et $x \in X$, $\lambda x.u$ est un terme qui représente la fonction qui à x associe l'objet décrit par la formule u :

- $x \in X$ et $u \in \lambda[X] \Rightarrow \lambda x.u \in \lambda[X]$.

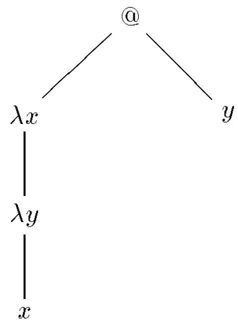
Il n'y a pas d'autres termes dans $\lambda[X]$.

Pour l'instant, les termes ne sont ni plus ni moins que des mots (*i.e.*des suites de symboles) écrits sur l'alphabet $X \cup \{(,),.,\lambda\}$.

La syntaxe

Substitutions dans les termes

Tout d'abord, les termes peuvent être représentés par des arbres. Par exemple au terme $(\lambda x.\lambda y.x)y$ on associe l'arbre suivant (dans lequel le symbole @ désigne l'opération binaire d'application) :



On dit qu'une occurence d'une variable x dans un terme u est libre si en remontant à la racine de l'arbre à partir de cet x on ne rencontre aucun λx. La notation $u[v/x]$ désigne alors le terme u dans lequel les occurences libres de x ont été remplacées par v. Il faut toutefois prendre garde au phénomène de “capture des variables” : considérons les termes $u = \lambda x.(xy)$ et $v = x$, on ne peut pas supposer que $u[v/y] = \lambda x.(xx)$ car, en utilisant la convention usuelle de renommage des variables (que l'on verra plus loin) u est équivalent à $\lambda z.(zy)$ et donc $u[v/y]$ doit être équivalent à $\lambda z.(zx)$. Cette convention de renommage des variables s'appelle alpha-conversion. En pratique, on l'applique sans la mentionner. Lors des substitutions, on s'arrangera à appliquer de telles conversions dans le but d'éviter les captures de variables.

Beta-réduction

Maintenant on peut appliquer la notion de substitution pour décrire comment opèrent les fonctions :

si u et v sont des termes et $x \in X$, alors les termes $(\lambda x.u)v$ et $u[v/x]$ sont équivalents, ce que l'on note :
– $(\lambda x.u)v \equiv u[v/x]$.

On prolonge cette relation de la manière suivante : si les termes u et u' sont équivalents, alors $(uv) \equiv (u'v)$,$(vu) \equiv (vu')$ et $\lambda x.u \equiv \lambda x.u'$ quels que soient $v \in \lambda[X]$ et $x \in X$. C'est en fait la clôture symétrique, réflexive et transitive de cette relation que l'on considèrera.

Quelques conventions

Une loi binaire, par exemple l'addition des entiers, peut être vue :

– d'une part comme l'application (à deux arguments) :

$$f: \begin{array}{rcl} \mathbf{N}^2 & \to & \mathbf{N} \\ (x,y) & \mapsto & x+y, \end{array}$$

– d'autre part comme l'application (à un argument) :

$$g: \begin{array}{rcl} \mathbf{N} & \to & \mathbf{N}^{\mathbf{N}} \\ x & \mapsto & \begin{array}{rcl} \mathbf{N} & \to & \mathbf{N} \\ y & \mapsto & x+y \end{array} \end{array}$$

le passage de f à g s'effectuant par une transformation évidente. La première forme est appelée forme curryfiée et la seconde forme décurryfiée. Comme on ne dispose pas (pas encore) de couples en lambda-calcul, c'est plutôt cette seconde forme que l'on va utiliser. Pour $x,y \in \mathbf{N}$, $x+y$ s'écrit alors $(g(x))(y)$ ce que l'on note plutôt $(gx)y$ ou encore gxy, l'application étant par convention associative à gauche : ce sera toujours le cas en lambda-calcul.

On dispose donc d'un langage $\lambda[X]$ muni d'une relation d'équivalence $\equiv$. Un programme (de même que ses données) seront des éléments de $\lambda[X]$ et exécuter le programme u sur les entrées $v_1, \ldots ,v_n$ consistera à (essayer de) trouver une forme normale (*i.e.*la plus simple possible) pour le terme $uv_1 \ldots v_n$. Avant de voir comment calculer de telles formes, on va construire quelques termes qui représentent la base des langages de programmation usuels.

La programmation

On va ici définir des termes particuliers qui simplifient la construction de programmes en lambda-calcul. Pour que ces termes puissent être ensuite utilisés à bon escient, on doit préciser quelles sont les propriétés qu'ils vérifient. Par exemple la terme «identité» défini par :

$$\mathtt{I} = \lambda x.x$$

possède la propriété suivante : quel que soit le terme $u \in \lambda[X]$, $\mathtt{I}u$ est équivalent à u. On notera une telle définition associée à l'énoncé d'une propriété sous la forme suivante :

$$\begin{array}{l} \mathtt{I} = \lambda x.x \\ [\![\forall u \in \lambda[X].\ \mathtt{I}u \equiv u]\!]. \end{array}$$

Les entiers

De manière générale, pour coder un objet mathématique en lambda-calcul, on se pose plutôt la question «à quoi ca sert ?» plutôt que «qu'est-ce que c'est ?» Par exemple les entiers servent (entre autres) à itérer un certain nombre de fois une fonction à partir d'une valeur initiale. 3 prend une fonction f et un argument x et renvoie la valeur $f(f(fx))$, donc le terme qui lui correspond est :

$$\overline{3} = \lambda f.\lambda x.f(f(fx)).$$

Plus généralement, un entier n est représenté par le terme :

$$\begin{array}{l} \overline{n} = \lambda f.\lambda x.\underbrace{f(\cdots(fx)\cdots)} \\ [\![\forall u,v \in \lambda[X].\ \forall n \in \mathbf{N}.\ \overline{n}\ uv \equiv \underbrace{u(\cdots(uv)\cdots)}]\!] \end{array}$$

(avec n occurences de f), en particulier $\overline{0} = \lambda f.\lambda x.x$. L'entier $\overline{n}$ itère n fois, son successeur doit itérer $n+1$ fois, il est par conséquent équivalent à :

$$\lambda f.\lambda x.f(\overline{n}fx)$$

d'où la forme fonctionnelle du successeur :

$$\begin{array}{l} \mathtt{succ} = \lambda n.\lambda f.\lambda x.f(nfx) \\ [\![\forall n \in \mathbf{N}.\ \mathtt{succ}\ \overline{n} \equiv \overline{n+1}]\!]. \end{array}$$

On obtient maintenant de manière immédiate les fonctions usuelles, par exemple $n+m$ se calcule en itérant n fois le successeur à partir de m, $\overline{n+m}$ doit donc être équivalent à $\overline{n}\ \mathtt{succ}\ \overline{m}$ ce qui nous donne la forme fonctionnelle de l'addition :

$$\begin{array}{l} \mathtt{plus} = \lambda n.\lambda m.n\ \mathtt{succ}\ m \\ [\![\forall n,m \in \mathbf{N}.\ \mathtt{plus}\ \overline{n}\ \overline{m} \equiv \overline{n+m}]\!]. \end{array}$$

En appliquant la même méthode, on réalise facilement les opérations de multiplication (`mul`) et d'exponentiation (`exp`).

Les booléens

Les booléens sont essentiellement employés en infomatique dans des expressions conditionnelles du style «si t alors u sinon w». Le booléen t peut alors être vu comme une fonction à deux arguments, *vrai* renvoyant son premier argument et *faux* renvoyant le second, d'où les définitions :

$$\texttt{vrai} = \lambda x.\lambda y.x \text{ et } \texttt{faux} = \lambda x.\lambda y.y.$$

On peut alors définir une forme fonctionnelle de l'expression conditionnelle en posant :

$$\begin{array}{l} \texttt{cond} = \lambda t.\lambda x.\lambda y.(txy) \\ \left[\!\!\left[\begin{array}{l} \forall u,v \in \lambda[X].\ \texttt{cond vrai}\ uv \equiv u \\ \forall u,v \in \lambda[X].\ \texttt{cond faux}\ uv \equiv v \end{array} \right]\!\!\right] \end{array}$$

On peut aussi réaliser les opérations booléennes usuelles au moyen de ces termes :

$$\begin{array}{l} \texttt{et} = \lambda x.\lambda y.\texttt{cond}\ xy\ \texttt{faux} \\ \left[\!\!\left[\begin{array}{l} \texttt{et vrai vrai} \equiv \texttt{vrai} \\ \texttt{et vrai faux} \equiv \texttt{faux} \\ \texttt{et faux vrai} \equiv \texttt{faux} \\ \texttt{et faux faux} \equiv \texttt{faux} \end{array} \right]\!\!\right] \end{array}$$

$$\begin{array}{l} \texttt{non} = \lambda x.\texttt{cond}\ x\texttt{faux vrai} \\ \left[\!\!\left[\begin{array}{l} \texttt{non vrai} \equiv \texttt{faux} \\ \texttt{non faux} \equiv \texttt{vrai} \end{array} \right]\!\!\right] \\ \ldots \end{array}$$

Profitons-en pour écrire une fonction testant si un entier est nul. On cherche en fait deux termes u et v tels que pour tout entier n on ait :

$$\begin{array}{l} \overline{n}uv \equiv \texttt{vrai} \text{ si } n = 0, \\ \overline{n}uv \equiv \texttt{faux} \text{ sinon,} \end{array}$$

or $\overline{0}uv \equiv v$, ce qui impose $v \equiv \texttt{vrai}$. Par ailleurs $\overline{1}uv \equiv (uv)$ et on doit donc avoir $(uv) \equiv \texttt{faux}$ ce qui motive pour u le choix du terme $u = \lambda x.\texttt{faux}$. On pose alors :

$$\begin{array}{l} \texttt{nul} = \lambda n.n(\lambda x.\texttt{faux})\texttt{vrai} \\ \left[\!\!\left[\begin{array}{l} \texttt{nul}\ \overline{0} \equiv \texttt{vrai} \\ \forall n \in \mathbf{N} \setminus \{0\}.\ \texttt{nul}\ \overline{n} \equiv \texttt{faux} \end{array} \right]\!\!\right] \end{array}$$

Couples et projections

Mathématiquement, un élément $(e,f) \in E \times F$ peut être vu comme une application d'un ensemble à deux éléments (par exemple $\{1,2\}$) dans $E \cup F$:

$$\begin{array}{rccl} c: & \{1,2\} & \rightarrow & E \cup F \\ & 1 & \mapsto & e \\ & 2 & \mapsto & f \end{array}$$

La première projection consiste à appliquer c en 1 et la seconde en 2. On va plutôt prendre $\{\texttt{vrai},\texttt{faux}\}$ comme ensemble à deux éléments. Si u et v sont deux termes, le terme suivant est une représentation du couple (u,v) :

$$\lambda t.\texttt{cond}\ tuv$$

En effet, appliqué en **vrai**, il renvoie u et en **faux**, il renvoie v. Par conséquent le terme suivant :

$$\texttt{cons} = \lambda x.\lambda y.\lambda t.\texttt{cond}\ txy$$

prend en arguments deux termes et renvoie le couple formé de ces deux termes. Les projections quant à elles s'écrivent :

$$\begin{array}{l} \texttt{p1} = \lambda x.x\ \texttt{vrai} \\ [\![\forall u,v \in \lambda[X].\ \texttt{p1}(\texttt{cons}\ uv) \equiv u]\!] \end{array}$$

$$\begin{array}{l} \texttt{p2} = \lambda x.x\ \texttt{faux} \\ [\![\forall u,v \in \lambda[X].\ \texttt{p2}(\texttt{cons}\ uv) \equiv v]\!] \end{array}$$

Comme premier exemple de programmation, voyons comment réaliser la fonction prédécesseur, notée **pred** telle que **pred** $\overline{n+1} \equiv \overline{n}$ pour tout entier n. Comme $\overline{n+1}$ itère $n+1$ fois une fonction, il faut trouver une fonction qui lors de sa première application «ne fait rien» et lors des suivantes prend le successeur :

$$\begin{array}{l} \texttt{aux} = \lambda x.\texttt{cond}(\texttt{p1}\ x) \\ \quad (\texttt{cons vrai}(\texttt{succ}(\texttt{p2}\ x))) \\ \quad (\texttt{cons vrai}(\texttt{p2}\ x)) \end{array}$$

appliquée en **cons faux** v (*i.e.*le couple $(\texttt{faux},v)$) elle renvoie $(\texttt{vrai},v)$ (elle ne fait donc rien sur la seconde composante) et appliquée en $(\texttt{vrai},v)$, elle renvoie $(\texttt{vrai},\texttt{succ}\ v)$. La construction du prédécesseur est alors immédiate en utilisant cette fonction auxiliaire :

$$\begin{array}{l} \texttt{pred} = \lambda n.\texttt{p2}(\overline{n}\ \texttt{aux}(\texttt{cons faux}\ \overline{0})) \\ [\![\forall n \in \mathbf{N}.\ \texttt{pred}\ \overline{n+1} \equiv \overline{n}]\!]. \end{array}$$

Remarquons qu'on ne donne aucune information sur le terme **pred** $\overline{0}$.

Récursivité

C'est la dernière construction à effectuer pour disposer d'un langage de programmation suffisament puissant. Etudions tout d'abord les points fixes : un point fixe pour un terme u est un terme v tel que $uv \equiv v$. Le résultat fondamental est que non seulement tous les termes possèdent un point fixe mais qu'en plus il existe un terme **fix** tel que pour tout terme u, **fix** u soit un point fixe pour u (*i.e.*: $u(\texttt{fix}\ u) \equiv \texttt{fix}\ u$). On peut trouver de nombreux termes réalisant un tel calcul de point fixe, le plus célèbre étant sans doute :

$$\texttt{fix} = \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$$

(combinateur de Curie). Pour tout $u \in \lambda[X]$ on a alors :

$$\begin{aligned} \texttt{fix}\ u &\equiv (\lambda x.u(xx))(\lambda x.u(xx)) \\ &\equiv u((\lambda x.u(xx))(\lambda x.u(xx))) \\ &\equiv u(\texttt{fix}\ u). \end{aligned}$$

La propriété que satisfait $\texttt{fix}$ est donc :

$$[\![\forall u \in \lambda[X]\ .u(\texttt{fix}\ u) \equiv u]\!].$$

Pour voir maintenant comment construire un opérateur de récursivité, étudions la fonction factorielle définie par :

$$\begin{cases} \mathrm{fact}(0) = 1 \\ \mathrm{fact}(n) = n \times (\mathrm{fact}(n-1)) \text{ si } n \geqslant 1. \end{cases}$$

Si f est une fonction de $\mathbf{N}$ dans $\mathbf{N}$, notons $\hat{f}$ la fonction définie par :

$$\begin{array}{rccl} \hat{f}: & \mathbf{N} & \to & \mathbf{N} \\ & 0 & \mapsto & 1 \\ & n & \mapsto & n \times f(n-1) \text{ si } n \geqslant 1. \end{array}$$

Soit Γ l'application de $\mathbf{N}^{\mathbf{N}}$ dans lui-même qui à f associe $\hat{f}$, fact est alors un point fixe de Γ (en fait, tout point fixe de Γ réalise la factorielle). Il suffit donc de trouver un terme u qui code la fonction Γ et $\texttt{fix}\ u$ sera la factorielle. Pour le terme u :

$$u = \lambda f.\lambda n.\texttt{cond}(\texttt{nul}\ n)\overline{1}(\texttt{mul}\ n(f(\texttt{pred}\ n)))$$

convient.

Essayons maintenant de généraliser : une définition par récurrence d'une fonction $f : \mathbf{N} \to E$ s'écrit :

$$\begin{cases} f(0) = x_0 \\ f(n) = g(n,f(n-1)) \text{ si } n \geqslant 1 \end{cases}$$

avec $x_0 \in E$ et $g : \mathbf{N}^2 \to \mathbf{N}$. La fonction Γ associée est alors :

$$\begin{array}{rccl} \Gamma: & \mathbf{N}^{\mathbf{N}} & \to & \mathbf{N}^{\mathbf{N}} \\ & f & \mapsto & \begin{array}{rcl} \mathbf{N} & \to & \mathbf{N} \\ 0 & \mapsto & x_0 \\ n \geqslant 1 & \mapsto & g(n,f(n-1)) \end{array} \end{array}$$

Le codage de cette fonction Γ est réalisé par le terme suivant :

$$\lambda f.\lambda n.\texttt{cond}(\texttt{nul}\ n)x_0(gn(f(\texttt{pred}\ n)))$$

d'où l'on déduit la forme fonctionnelle de Γ (prenant en argument x_0 et g) :

$$\begin{array}{l} \texttt{gamma} = \\ \quad \lambda x.\lambda g.\lambda f.\lambda n. \\ \qquad \texttt{cond}(\texttt{nul}\ n) \\ \qquad\quad x \\ \qquad\quad (gn(f(\texttt{pred}\ n))). \end{array}$$

Le terme suivant est donc un opérateur de récursivité :

$$\texttt{rec} = \lambda x.\lambda g.\texttt{fix}(\texttt{gamma}\ xg)$$
$$\left[\!\!\left[\begin{array}{l} \texttt{rec}\ uv\overline{0} \equiv u \\ \forall n \in \mathbf{N}.\ \texttt{rec}\ \overline{n+1}\ uv \equiv v\ \overline{n+1}\ (\texttt{rec}\ uv\overline{n}) \end{array}\right]\!\!\right]$$

appliqué à des termes x_0 et g qui définissent une fonction f, $\texttt{rec}\ x_0 g$ définit un terme codant f.

Un exemple : reconnaissance du langage de Dick

Nous allons maintenant donner un véritable exemple de programme réalisé en lambda-calcul : on va chercher un terme $u \in \lambda[X]$ tel que $uw \equiv \texttt{vrai}$ si w est un terme représentant un mot bien parenthésé et $uw \equiv \texttt{faux}$ si w est un terme représentant un mot mais qui n'est pas bien parenthésé. Avant d'en arriver à cette construction, réexpliquons brièvement ce que sont les mots et le langage de Dick.

Mots sur un alphabet fini ou dénombrable

Soit A un ensemble (fini ou dénombrable) qu'on appellera alphabet. L'ensemble des mots construits sur A (noté A^*) est tout simplement l'ensemble les suites finies (eventuellement vides) d'éléments de A. On peut munir A d'une structure de monoïde avec la loi de composition interne :

$$(a_1,\cdots,a_p).(b_1,\cdots,b_q) = (a_1,\cdots,a_p,b_1,\cdots,b_q).$$

Dans la suite, $(a_1,\cdots,a_p)$ sera noté plus simplement $a_1\cdots a_p$. On considère ici l'alphabet à deux lettres $A = \{[,]\}$ et on définit $p : A^* \to \mathbf{Z}$ comme étant l'unique morphisme de monoïdes tel que $p([) = +1$ et $p(]) = -1$. On dit qu'un mot $w = a_1\cdots a_n \in A^*$ est un mot de Dick (ou encore «bien parenthésé») si $p(a_1\cdots a_i) \geqslant 0$ pour tout $i \in [\![1,n]\!]$ et $p(w) = 0$ (le lecteur pourra vérifier qu'il s'agit bien là des mots correspondant à la notion usuelle de «bon parenthésage»). Comme on a introduit de nouveaux objets mathématiques (entre autres les mots et les entiers relatifs), on doit tout d'abord chercher comment les représenter en lambda-calcul avant de passer à la phase de reconnaissance. Afin d'alléger les notations, on fera l'abus d'écriture suivant : si $u,v \in \lambda[X]$, la notation (u,v) désignera le terme $\texttt{cons}\ uv$.

Les entiers relatifs

Nous n'aurons besoin que de deux opérations sur les entiers relatifs : un calcul de successeur et un calcul de prédécesseur. Il faudra aussi prévoir un terme pour tester la positivité et un pour la nullité. De manière générale, pour $n \in \mathbf{N}$ on notera $\widetilde{n} = (\mathtt{vrai},\overline{n})$ et $\widetilde{-n} = (\mathtt{faux},\overline{n})$. Remarquons qu'on a défini $\widetilde{0}$ de deux manières différentes. Auparavant on avait associé à chaque entier (naturel) un terme et ceci de manière unique. Ici chaque entier (relatif) possède un ensemble de représentants. Maintenant, lorsqu'on écrira $u = \widetilde{n}$ (ou $u \equiv \widetilde{n}$), cela voudra dire «u est un représentant de $n \in \mathbf{Z}$» (ou «u est équivalent à un représentant de $n \in \mathbf{Z}$») et une propriété du type $u\widetilde{n} \equiv \widetilde{m}$ devra se lire "si t est un représentant de $n \in \mathbf{Z}$, alors ut est équivalent à un représentant de $m \in \mathbf{Z}$". Les mêmes conventions auront cours lorsque, plus loin, on définira ce qu'est un représentant d'un mot $w \in A^*$. Le calcul du successeur se fait alors sans difficulté. Pour plus de commodité, on définit également un terme qui prend l'opposé :

$$
\begin{array}{l}
\mathtt{succz} = \\
\quad \lambda n. \\
\qquad \mathtt{cond}(\mathtt{p1}\ n) \\
\qquad\quad (\mathtt{vrai},(\mathtt{succ}\ (\mathtt{p2}\ n))) \\
\qquad\quad (\mathtt{cond}(\mathtt{nul}(\mathtt{p2}\ n)) \\
\qquad\qquad (\mathtt{vrai},\overline{1}) \\
\qquad\qquad (\mathtt{faux},(\mathtt{pred}(\mathtt{p2}\ n)))) \\
[\![\forall n \in \mathbf{Z}.\ \mathtt{succz}\ \widetilde{n} \equiv \widetilde{n+1}]\!]
\end{array}
$$

$$
\begin{array}{l}
\mathtt{oppz} = \\
\quad \lambda n. \\
\qquad ((\mathtt{non}(\mathtt{p1}\ n)),(\mathtt{p2}\ n)) \\
[\![\forall n \in \mathbf{Z}.\ \mathtt{oppz}\ \widetilde{n} \equiv \widetilde{-n}]\!]
\end{array}
$$

$$
\begin{array}{l}
\mathtt{predz} = \\
\quad \lambda n. \\
\qquad \mathtt{oppz}(\mathtt{succz}(\mathtt{oppz}\ n)) \\
[\![\forall n \in \mathbf{Z}.\ \mathtt{predz}\ \widetilde{n} \equiv \widetilde{n-1}]\!]
\end{array}
$$

$$
\begin{array}{l}
\mathtt{positz} = \\
\quad \lambda n. \\
\qquad \mathtt{ou}(\mathtt{p1}\ n)(\mathtt{nul}(\mathtt{p2}\ n)) \\
\left[\!\!\left[\begin{array}{l} \forall n \in \mathbf{N}.\ \mathtt{positz}\ \widetilde{n} \equiv \mathtt{vrai} \\ \forall n \in \mathbf{N} \setminus \{0\}.\ \mathtt{positz}\ \widetilde{-n} \equiv \mathtt{faux} \end{array} \right]\!\!\right]
\end{array}
$$

Comparaison et moins

Le plus simple pour comparer deux entiers (naturels) n et m consiste à convertir m en entier relatif, iérer n fois le prédécesseur et vérifier si le résultat est positif. Cela s'écrit :

$$
\begin{array}{l}
\mathtt{inf} = \\
\quad \lambda n.\lambda m. \\
\qquad \mathtt{positz}(n\ \mathtt{predz}(\mathtt{vrai},m)) \\
\left[\!\!\left[\begin{array}{l} \forall n,m \in \mathbf{N}.\ n \leqslant m \Rightarrow \mathtt{inf}\ \overline{n}\ \overline{m} \equiv \mathtt{vrai} \\ \forall n,m \in \mathbf{N}.\ n > m \Rightarrow \mathtt{inf}\ \overline{n}\ \overline{m} \equiv \mathtt{faux} \end{array} \right]\!\!\right]
\end{array}
$$

On peut aussi construire la fonction **moins** qui retire m à n (à condition que n soit plus grand que m) :

$$
\begin{array}{l}
\mathtt{moins} = \\
\quad \lambda n.\lambda m. \\
\qquad m\ \mathtt{pred}\ n \\
[\![\forall n,m \in \mathbf{N}.\ \mathtt{moins}\ \overline{n+m}\ \overline{m} \equiv \overline{n}]\!].
\end{array}
$$

Mots

Comme on l'a vu, l'alphabet sur lequel les mots sont construits est supposé dénombrable, on peut donc supposer par exemple que $A = \mathbf{N}$. Un mot $w = a_1 \cdots a_n$ sera représenté par un terme t tel que $t\overline{0} \equiv \overline{n}$ et $t\overline{i} \equiv \overline{a_i}$ pour $i \in [\![1,n]\!]$. On dira qu'un tel terme t est un représentant de w ce que l'on notera $t = \overline{w}$ (et qu'on devrait en toute rigueur noter $t \in \overline{w}$). La fonction donnant la longueur d'un mot se construit alors de manière triviale :

$$
\begin{array}{l}
\mathtt{long} = \lambda w.w\overline{0} \\
[\![\forall w = a_1 \ldots a_n \in A^*.\ \mathtt{long}\ \overline{w} \equiv \overline{n}]\!]
\end{array}
$$

de même que la fonction renvoyant la $i^{\text{ème}}$ lettre :

$$
\begin{array}{l}
\mathtt{lettre} = \lambda w.\lambda i.wi \\
\left[\!\!\left[\begin{array}{l} \forall w = a_1 \ldots a_n \in A^*.\ \forall i \in \{1, \ldots ,n\}. \\ \qquad \mathtt{lettre}\ \overline{w}\ \overline{i} \equiv \overline{a_i} \end{array} \right]\!\!\right]
\end{array}
$$

La fonction réalisant la concaténation de deux mots est plus compliquée à définir, le lecteur pourra vérifier (à titre d'exercice) que le terme suivant convient :

$$
\begin{array}{l}
\mathtt{concat} = \\
\quad \lambda u.\lambda v.\lambda n. \\
\qquad \mathtt{cond}(\mathtt{nul}\ n) \\
\qquad\quad (\mathtt{plus}\ (\mathtt{long}\ u)(\mathtt{long}\ v)) \\
\qquad\quad (\mathtt{cond}(\mathtt{inf}\ n\ (\mathtt{long}\ u)) \\
\qquad\qquad (un) \\
\qquad\qquad (v\ (\mathtt{moins}\ n\ (\mathtt{long}\ u)))) \\
[\![\forall w_1,w_2 \in A^*.\ \mathtt{concat}\ \overline{w_1}\ \overline{w_2} \equiv \overline{w_1 w_2}]\!].
\end{array}
$$

Finalement, prenant en argument un entier a, le terme suivant construit le mot à une lettre égal à a :

$$
\begin{array}{l}
\mathtt{base} = \lambda a.\lambda n.\mathtt{cond}\ (\mathtt{nul}\ n)\ \overline{1}\ a \\
[\![\forall a \in A.\ \mathtt{base}\ \overline{a} \equiv \overline{a}]\!].
\end{array}
$$

Mots de Dick

Fixons $w = a_1 \dots a_n \in A^*$, on peut alors associer à w la fonction :

$$\begin{array}{rccl} p_w : & \{0, \dots, n\} & \to & \mathbf{Z} \\ & i & \mapsto & p(a_1 \dots a_i) \end{array}$$

(avec par convention $p_w(0) = 0$) de sorte que $p(w) = p_w(n)$. Cette fonction associée à w possède une définition récursive naturelle :

$$\begin{cases} p_w(0) = 0 \\ p_w(i) = p_w(i-1) + 1 \text{ si } i \in \{1, \dots, n\} \text{ et } a_i = [\\ p_w(i) = p_w(i-1) - 1 \text{ si } i \in \{1, \dots, n\} \text{ et } a_i =] \end{cases}$$

Identifions l'alphabet $A = \{[,]\}$ avec $\{0,1\}$, on peut alors représenter p_w par le terme suivant :

$$\begin{array}{l} \texttt{rec}\ \widetilde{0}\ (\lambda n.\lambda z. \\ \quad \texttt{cond}\ (\texttt{nul}\ (\texttt{lettre}\ wn)) \\ \qquad (\texttt{succ}\ z) \\ \qquad (\texttt{pred}\ z)) \end{array}$$

et si on abstrait ce terme en fonction de w on obtient :

$$\begin{array}{l} \texttt{p} = \\ \quad \lambda w.\texttt{rec}\ \widetilde{0}\ (\lambda n.\lambda z. \\ \qquad \texttt{cond}\ (\texttt{nul}\ (\texttt{lettre}\ wn)) \\ \qquad\quad (\texttt{succ}\ z) \\ \qquad\quad (\texttt{pred}\ z)) \end{array}$$

$$\left[\!\!\left[\begin{array}{l} \forall w = a_1 \dots a_n \in A^*.\ \forall i \in \{1, \dots, n\}. \\ \quad \texttt{p}\ \underline{w}\ \underline{i} \equiv \overline{p(a_1 \dots a_i)} \end{array} \right]\!\!\right]$$

Posons finalement :

$$\begin{array}{rl} D' = & \{w = a_1 \dots a_n \in A^* | \\ & \quad \forall i \in \{1, \dots, n\}.p(a_1 \dots a_i) \geqslant 0\} \end{array}$$

on a alors $D = D' \cap \{w \in A^* | p(w) = 0\}$. Le terme suivant permet de tester si un mot est dans D' :

$$\begin{array}{l} \texttt{testDprime} = \lambda w.\texttt{rec}\ \texttt{vrai} \\ \quad (\lambda n.\lambda z.\texttt{et}\ z\ (\texttt{positz}\ (\texttt{p}\ wn))) \end{array}$$

et donc on en déduit le test d'appartenance à D :

$$\begin{array}{l} \texttt{testD} = \lambda w.\texttt{et} \\ \quad (\texttt{testDprime}\ w\ (\texttt{long}\ w)) \\ \quad (\texttt{nulz}\ (\texttt{p}\ w\ (\texttt{long}\ w))). \end{array}$$

Questions de stratégies

Lorsu'on cherche une forme normale pour un terme, il est commode de remplacer la relation d'équivalence $\equiv$ par une relation de réduction (ou règle de réecriture) que l'on appelle beta-réduction et définie par :

- $(\lambda x.u)v \to u[v/x]$,
- si $u \to u'$, alors $(uv) \to (u'v)$, $(vu) \to (vu')$ et $\lambda x.u \to \lambda x.u'$ quels que soient v et x.

On cherche alors à réduire successivement les termes à l'aide de cette relation pour obtenir une forme irréductible. Deux problèmes peuvent alors se poser. D'une part certains termes n'ont pas de forme normale, par exemple :

$$u = (\lambda x.(xx))(\lambda x.(xx))$$

peut se réduire indéfiniment sur lui-même. Par ailleurs, si on pose $v = \lambda x.y$, alors on peut réduire indéfiniment vu sur lui-même (si on choisit de réduire d'abord dans u) ou bien aboutir immédiatement à la forme normale y si on subsitue u à x dans y. En fait ce principe est général : si dans un terme, on réduit toujours le lambda qui se trouve le plus à gauche, alors on aboutira à une forme normale (si toutefois le terme en possède une).

Pour en savoir plus...

[B] H.P. Barendregt, *The lambda calculus, its syntax and semantics*, Studies in logic and the foundations of mathematics.

[L] R. Lalement, *Logique, réduction, résolution*, Masson.

Petits problèmes

Nous vous proposons ici quelques problèmes que nous avons rencontrés, et pour lesquels, en général, nous n'avons pas de résultat satisfaisant. Vous pouvez nous soumettre vos résultats à l'adresse du journal : `hubris@rip.ens-cachan.fr`, afin que nous les publiions dans les numéros suivants.

Le problème des timbres (bis)

On considère un ensemble de timbres, dont les valeurs se situent dans un ensemble de k entiers premiers entre eux, et dont le nombre est illimité pour chaque valeur. On peut alors montrer qu'à partir d'une certain entier, toutes les valeurs sont réalisables. Le problème est de le montrer, mais surtout d'exprimer l'entier minimal à partir duquel toutes les valeurs sont réalisables.

Par exemple, si on a un ensemble infini de timbres de trois francs et un ensemble infini de timbres de cinq francs, alors on peut réaliser toutes les valeurs à partir de huit francs. Mais sept francs n'est pas réalisable.

Nous allons montrer que, si A_m est le nombre de solutions entières positives de l'équation diophantienne $px+qy=m$, alors A_m est supérieur ou égal à un dès que m est supérieur ou égal $r=(p-1)(q-1)$. Nous montrerons également que la valeur $r-1$ n'est pas réalisable, ce qui répondra complètement au problème.

Pour montrer ces résultats, nous allons utiliser la théorie des séries génératrices, et en particulier le fait que deux séries formelles sont égales si, et seulement si, tous les termes de la suite la définissant sont égaux.

Posons tout d'abord

$$\begin{aligned}\Phi(X) &= \frac{(1-X)(1-X^{pq})}{(1-X^p)(1-X^q)}\\ &= \frac{1+X^p+\cdots+X^{p(q-1)}}{1+X+\cdots+X^{q-1}},\end{aligned}$$

et montrons que Φ est en fait un polynôme : pour le voir, nous utilisons la décomposition de la fraction rationnelle Φ en éléments simples :

$$\begin{aligned}\Phi(X) = E(X) &+ \frac{a}{(1-X)^2} + \frac{b}{1-X}\\ &+ \sum_{i=1}^{p-1}\frac{\alpha_i}{\zeta_p^i - X} + \sum_{j=1}^{q-1}\frac{\beta_j}{\zeta_q^j - X}.\end{aligned}$$

En multipliant les deux membres de l'égalité par $(1-X)^2$, puis en prenant $X=1$, on vérifie que a est nul. De la même façon, b est nul, ainsi que les α_i et les β_j, car p et q sont premiers entre eux, et les racines de l'unité sont toutes différentes. Il en résulte que Φ est un polynôme. Son degré vaut $(p-1)(q-1)$. On note Φ_l le coefficient du terme X^l de Φ.

On a alors formellement

$$\begin{aligned}\frac{1}{(1-X^p)(1-X^q)} &= \left(\sum_{k\geqslant 0} X^{kp}\right)\left(\sum_{l\geqslant 0} X^{lq}\right)\\ &= \sum_{k,l\geqslant 0} X^{kp+lq}\\ &= \sum_{m\geqslant 0} A_m X^m.\end{aligned}$$

De même, on a l'égalité suivante :

$$\frac{1}{(1-X)(1-X^{pq})} = \sum_{m\geqslant 0} B_m X^m$$

où B_m représente le nombre de solutions de l'équation $x+pqy=m$. Cette équation est plus simple que la première, et le nombre de ses solutions entières positives est facilement calculable : on a en effet

$$B_m = \left\lfloor \frac{m}{pq} \right\rfloor + 1$$

puisque c'est le nombre de valeurs que peut prendre y pour que pqy reste inférieur à m. On vérifie alors en particulier que $B_{m+pq} = B_m + 1$.

Par définition de A_m, B_m et Φ, on a

$$\begin{aligned}\sum_{m\geqslant 0} A_m X^m &= \left(\sum_{l\geqslant 0}\Phi_l X^l\right)\left(\sum_{k\geqslant 0} B_k X^k\right)\\ &= \sum_{m\geqslant 0}\left(\sum_{l\geqslant 0}\Phi_l B_{m-l}\right) X^m.\end{aligned}$$

On en déduit, au vu de la remarque initiale, que

$$A_m = \sum_{l=0}^{m}\Phi_l B_{m-l}.$$

Alors,

- on a l'égalité

$$\begin{aligned}A_{m+pq} &= \sum_{i=0}^{m+pq}\Phi_i(B_{m-i}+1)\\ &= A_m + \sum_{i=0}^{m+pq}\Phi_i.\end{aligned}$$

Donc, comme $m+pq$ est supérieur au degré de Φ, on en déduit $A_{m+pq} = A_m + \Phi(1) = A_m + 1$.

– si $m < pq$, on a $B_m = 1$, et $A_m = \sum_{k=1}^{s} \Phi_k$, où s est le plus petit de m et du degré de Φ, *i.e.*, $(p-1)(q-1)$. Donc, si m est compris entre r et $pq-1$, on a $A_m = \Phi(1) = 1$.

Prenons maintenant m strictement inférieur à r. Supposons que le problème ait deux solutions (s,t) et (s',t') distinctes. Alors dans ce cas,

$$(s-s')p = (t-t')q$$

et p divisera $t-t'$. En particulier, p sera inférieur à t, et $m = sp + tq$ sera supérieur à pq, ce qui est exclu. Donc, si m est strictement inférieur à r, A_m est soit nul, soit égal à 1. Par ailleurs, toujours pour ces choix de m, on a $A_m = \sum_{k=0}^{m} \Phi_k$, donc $\Phi_i = A_i - A_{i-1}$. Donc, pour tout i, $\Phi_i \in \{-1,0,1\}$. De plus, si $\Phi_i = \Phi_{i+1}$, alors ils sont nuls, car sinon A_{i+1} sera supérieur ou égal à deux. Enfin, comme Φ_r vaut 1 et A_r également, on a $A_{r-1} = 0$.

Nous venons de montrer que r est le plus petit entier à partir duquel tous les entiers sont « accessibles ».

Ne nous arrêtons pas en si bon chemin, et montrons que la moitié des entiers strictement inférieurs à r sont accessibles. On vérifie que :

$$\begin{aligned}\sum_{m=0}^{r} A_m &= (r+1)\Phi_0 + r\Phi_1 + \cdots + 2\Phi_{r-1} + \Phi_r \\ &= \sum_{m=0}^{r} (r+1-i)\Phi_i \\ &= (r+1)\Phi(1) - \Phi'(1) \\ &= (r+1) + \Phi'(1).\end{aligned}$$

Or la dérivée logarithmique de Φ donne

$$\begin{aligned}\frac{\Phi'(X)}{\Phi(X)} = &-\frac{1+2X+\cdots+(q-1)X^{q-2}}{1+X+\cdots+X^{q-1}} \\ &+ \frac{pX^{p-1}+2pX^{2p-1}+\cdots+(q-1)pX^{(q-1)p-1}}{1+X^p+\cdots+X^{p(q-1)}}.\end{aligned}$$

D'où $\Phi'(1) = \frac{1}{2}(p-1)(q-1)$. Comme A_m est inférieur ou égal à 1 si $m < pq$, on obtient

$$\sum_{m=0}^{r} A_m = \frac{r}{2} + 1$$

et comme A_r vaut 1, on obtient le résultat annoncé.

N.M.

Livres

Word processing and group theory,
D. Epstein et al.,
Jones and Bartlet publishers, 1992

Les descriptions usuelles des groupes ne sont pas toujours aussi utilies qu'on pourrait le vouloir ; en particulier, elles ne sont pas toujours algoritihmiques. Par exemple, quand un groupe est donné par générateurs et relations, il n'existe pas d'algorithme permettant de dire si deux mots correspondent au meme élément.

Néanmoins, les groupes que l'on rencontre proviennent souvent de situations géométriques, ce peuvent etre par exemple des groupes fondamentaux de variétés riemanniennes. On peut donc espérer que cette origine géométrique entraine des propriétés supplémentaires sur le groupe, qui permettent de mieux le visualiser et de mieux calculer avec.

Rappelons ce qu'est un *graphe de Cayley.* Si G est un groupe et S un système (fini) de générateurs on considère le graphe orienté dont les sommets sont les éléments du groupe et dont les arrètes sont les $g \longrightarrow sg$ pour tout $s \in S$ et $g \in G$. On en fait un espace métrique en décrétant que les arrètes sont de longueur un. Si G provient d'une situation géométrique, par exemple, si c'est le groupe fondamental d'une variété hyperbolique compacte, la géométrie du graphe de Cayley peut être vue comme un modèle combinatoire de cette situation.

Il existe alors (souvent) un automate fini qui nous dit si deux mots sur S correspondent au même élément de G (je simplifie un peu). On parle alors de *groupe automatique.*

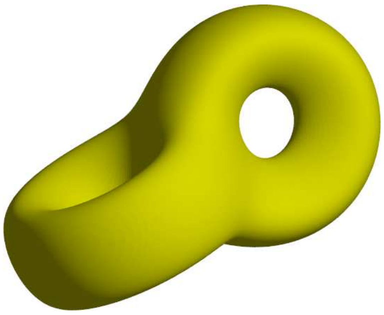
Comme exemples, on peut citer les groupes finis, les groupes abéliens de type fini, les groupes de tresses, beaucoup de groupes hyperboliques et beaucoup de groupes de Coxeter.

Ce livre présenta les fondations du sujet, par ses initiateurs. Pour une autre présentation du sujet, plus succinte, ainsi qu'une liste d'applications concrètes (en image de synthèse, par exemple), on peut consulter *Automatic groups, a guided tour,* L'enseignement mathématique 32 (1992) p. 291–313. **V.Z.**

Algorithms and computer methods for 3-manifolds,
A.T. Fomenko and S.V. Matveev, Kluwer, 1997

Contrairement à ce que laisse suggérer le titre, ce livre ne parle pas du tout d'informatique (ce sera le volume suivant), mais uniquement de la géométrie des variétés de dimensions deux et trois. Les dessins de Fomenko soulignent la nature profondément visuelle du sujet.

Le livre commence par l'étude des surfaces : les auteurs démontrent par exemple avec force dessins que toute surface compacte orientée est un « tore » à plusieurs trous.



On peut ensuite s'intéresser au groupe des homéomorphismes d'une surface, ou plutôt, pour obtenir un objet de taille plus raisonnable, les homéomorphismes à déformation près. On peut voir que ce groupe en engendré par les *torsions de Dehn*, homéomorrphismes qui consistent à découper la surface le long d'une courbe fermée simple, faire un tour complet et recoller. (On connait d'ailleurs une présentation *finie* de ce groupe).

Une manière de construire une variété de genre trois est la suivante : prendre deux tores solides (de même genre) et les recoller le long d'un homéomorphisme de leur bord. On peut montrer que toute variété de dimension trois peut s'obtenir de cette manière : on parle de décomposition de *Heegaard.* Décrire une telle décomposition revient à décrire l'homéomorphisme de la surface utilisé pour le recollement (modulo déformation).

On peut se demander si deux décompositions de Heegaard correspondent à des variétés isomorphes. On ne dispose que d'une réponse partielle : en genre deux (c'est le genre du bord des tores solides), on sait reconnaître la sphère de dimension trois.

On peut montrer l'existence et l'unicité de la décomposition en somme connexe de variétés « premières » (c'est la même définition que pour les nombres premiers : non décomposables en somme connexe de variétés non triviales, *i.e.*, non homéomorphes à la sphère)

Après un chapitre sur la théorie des nœuds et leurs invariants, le livre démontre que toute variété de dimension trois peut s'obtenir à partir de la sphère $\mathbf{S}^3$ par *chirurgie* le long d'un lien, *i.e.*, en découpant un voisinage tubulaire d'un lien (plusieurs nœud, éventuellement emmêlés) et en le recollant après avoir fait subir un homéomorphisme à son bord. On sait dire quaud deux liens donnent la même variété (quand ils sont reliés par des *mouvements de Kirby*). Dans certains cas, les invariants de nœuds permettent de définir des invariants de variétés de dimension trois.

Le livre s'achève sur l'étude de certaines classes de variétés de dimension trois, comme les variétés de Seifert. **V.Z.**