Groupe de travail pour élèves de lycée

Bons ordres sur $\mathbb{N}$

par

Xavier Caruso

Le 27 avril 2003

Table des matières

1 Les notions d'ordre et d'ordinaux

1.1 Définitions et exemples

Un *ordre* sur un ensemble E est une partie de $E \times E$, que l'on note généralement $\leqslant$ et qui vérifie les deux conditions suivantes (on note $x \leqslant y$ pour $(x, y) \in \leqslant$) :

i) $x \leqslant x$

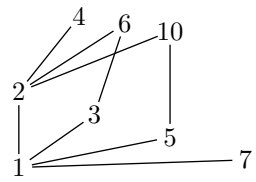
ii) si $x \leqslant y$ et $y \leqslant z$, alors $x \leqslant z$

En fait, ce que l'on vient de définir est un *pré-ordre*. Pour un véritable ordre, il faut ajouter la condition supplémentaire :

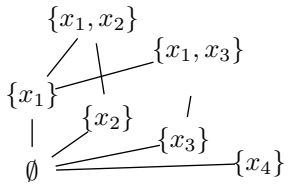
iii) si $x \leqslant y$ et $y \leqslant x$, alors $x = y$

qui n'est toutefois pas primordiale comme nous allons le constater au fil de cet article.

Notons qu'étant donnés deux éléments x et y de E, on n'a pas forcément $x \leqslant y$ ou $y \leqslant x$. Ceci est en général vrai pour les ordres classiques, mais il en est d'autres, pas moins naturels, pour lesquels cet axiome n'est pas vérifié. Un premier exemple est celui de la divisibilité sur $\mathbb{N}$ (ou sur $\mathbb{Z}$ mais dans ce cas, la troisième condition n'est plus vérifiée) : 2 et 3 sont alors incomparables. On peut représenter cet ordre par le diagramme suivant, pas forcément très explicite :



Un autre exemple est celui de l'inclusion sur $\mathcal{P}(E)$, E étant un ensemble quelconque fixé. Là encore, on peut faire un diagramme qui ressemble beaucoup au précédent. Cela donne :



Les deux ordres précédents ont beaucoup de points en commun. Par exemple, ils ont tous les deux un plus petit et un plus grand élément : pour la divisibilité, le plus petit élément est 1, le plus grand est 0 ; pour l'inclusion, le plus petit élément est l'ensemble vide, le plus grand est E lui-même. D'autre part, si l'on prend deux éléments quelconques, disons X et Y, on ne peut pas toujours les comparer mais on peut, parmi ceux qui sont à la fois plus grands (resp. plus petits) que X et Y, en trouver un plus petit (resp. plus grand). Pour la divisibilité, par exemple, être à la fois plus grand que X et Y signifie simplement être un multiple de X et de Y c'est-à-dire un multiple de leur « plus petit commun multiple ». C'est ce nombre le plus petit majorant. De même pour les sous-ensembles de E, être plus petit que X et Y, c'est être inclus dans X et Y *i.e.* dans leur intersection. Cette intersection est bien le plus grand des minorants communs.

On peut se poser cette question de façon générale : étant donnés un ensemble E muni d'un ordre et une partie A de E, existe-il forcément un $x \in E$ plus grand que tous les

éléments de A. Et le cas échéant, parmi tous ces x convenables, y en a-t-il un plus petit ? La réponse est en général négative aux deux questions ; on peut par exemple prendre $E = \mathbb{R}$ et $A = \mathbb{N}$ pour s'en convaincre. Il est aussi en général faux qu'une réponse affirmative à la première question entraîne une réponse affirmative à la seconde. Il est cependant un peu plus délicat de construire un contre-exemple principalement parce qu'il est théorème qui affirme qu'un tel contre-exemple ne peut exister avec $E = \mathbb{R}$. Toutefois, il suffit de considérer $E = \mathbb{R}^\star$ l'ensemble des réels non nuls et $A = \mathbb{R}_-^\star$ l'ensemble des réels strictement négatifs.

Toutefois, on introduit la terminologie suivante. Si la réponse à la première question est affirmative, la partie A est dite *majorée* dans E. Et dans ce cas, si la réponse à la deuxième question est affirmative, le plus petit élément dont on parle est appelé la *borne supérieure* de A dans E. Remarquons que l'on a toujours supposé implicitement l'unicité des plus petits et des plus grands éléments. Cela est vrai pour un ordre : en effet, si x et y sont deux plus petits éléments par exemple, on a forcément $x \leqslant y$ et $y \leqslant x$ et donc $x = y$ par la troisième propriété. Toutefois, cela n'est plus forcément vrai pour un pré-ordre, on prendra soin dans ce cas de parler d'*un* plus petit élément, d'*un* plus grand élément et d'*une* borne supérieure.

Il est encore possible de donner de multiples définitions et de propriétés générales sur les ordres, mais rien de plus nous sera utile pour l'exposé que l'on souhaite faire.

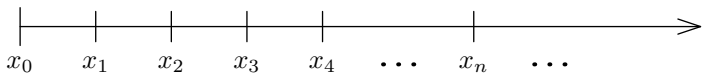
1.2 Les bons ordres

Considérons E un ensemble muni d'une relation d'ordre notée $\leqslant$. On dit que $\leqslant$ est un *bon ordre* si toute partie non vide de E admet un plus petit élément.

Décortiquons la définition qui précède et considérons pour cela un ensemble E *bien ordonné* (*i.e.* muni d'un bon ordre). Commençons avant tout par faire une remarque : si x et y sont deux éléments de E, la paire $\{x, y\}$ est non vide et incluse dans E et admet donc un plus petit élément. Celui-ci ne peut être que x ou y. Dans le premier cas, on obtient $x \leqslant y$, et dans le second $y \leqslant x$. En résumé, on vient de prouver que les bons ordres ont une propriété agérable, à savoir que deux éléments sont toujours comparables. En particulier, les diagrammes que l'on peut faire pour les représenter ne contiennent pas de ramification. C'est pour cette raison que l'on dit parfois dans ce cas que l'ordre est *linéaire* (ou encore *total*).

Forts de cette remarque, voyons maintenant plus précisément la structure de E. Si E est vide, il n'y a pas grand chose à dire. Supposons donc que ce ne soit pas le cas. Il admet alors un plus petit élément que l'on note x_0. Si la partie $E \setminus \{x_0\}$ (E privé de l'élément x_0) est vide, E était réduit à x_0 et il n'y a plus grand chose à dire. Sinon, cette partie admet à son tour un plus petit élément que l'on note x_1. On continue ainsi de suite en regardant désormais $E \setminus \{x_0, x_1\}$.

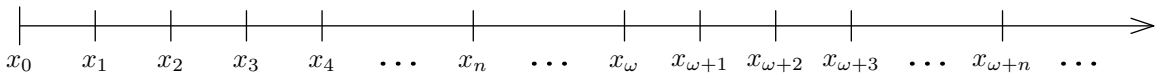
Ainsi, on démontre que si E n'est pas fini, on peut commencer à le représenter de la façon suivante :



Autrement dit, si E est infini, E « commence comme » l'ensemble des entiers naturels. Cependant, il n'est pas exclu que E ne continue pas. Regardons la partie formée de tous

les x_i pour $i \in \mathbb{N}$ et notons A son complémentaire. S'il est vide, l'ensemble E s'identifie à l'ensemble des entiers naturels après avoir renommé les éléments. Si A n'est pas vide, il admet un plus petit élément que l'on pourrait noter x_∞, mais que l'on note x_ω[1].

Et cela continue encore : si on n'a pas tout épuisé, on peut enlever à E tous les x_i pour i entier *et* x_ω. Cette partie est alors non vide, elle admet un plus petit élément que l'on appelle $x_{\omega+1}$. Remarquons que ceci est en fait très général : chaque fois qu'il existe un x_α, il va possiblement exister un $x_{\alpha+1}$[2]. On obtient au final (si E n'a pas été épuisé d'ici là) la représentation suivante :



Mais encore une fois, cela ne s'arrête ici : s'il reste des éléments dans E, il y en a parmi eux un plus petit, disons $x_{\omega 2}$. Puis vient $x_{\omega 2+1}$ et ainsi de suite. On se rend compte que l'on peut continuer ce jeu très longtemps et que l'on risque de tomber à court de notation avant d'avoir sorti un par un tous les éléments de E (typiquement si E est indénombrable). Mais peu importe, essayons quand même de fixer les noms des premiers indices.

1.3 Les premiers ordinaux

De la même façon que les « premiers » indices sont ce que l'on appelle communément les entiers, les indices en général sont les *ordinaux*. Il y a une définition rigoureuse des ordinaux mais ce n'est vraiment pas utile de la connaître au même titre que c'est totalement inutile (si ce n'est nuisible) de connaître une définition rigoureuse des entiers pour arriver à les manipuler. Il est toutefois bon de bien remarquer que les ordinaux sont naturellement ordonnés et même bien ordonnés. Décrivons à présent le début du commencement de l'ensemble des ordinaux.

Il suffit de constater deux choses : d'une part, comme nous l'avons déjà dit, chaque fois que l'on a un ordinal α, il y a un ordinal successeur noté $\alpha+1$ et d'autre part tout ensemble d'ordinaux admet une borne supérieure. Ces remarques faites, on peut commencer à dresser le tableau suivant :

$$\begin{array}{cccccccc}
0 & 1 & 2 & 3 & 4 & 5 & \dots \\
\omega & \omega+1 & \omega+2 & \omega+3 & \omega+4 & \omega+5 & \dots \\
\omega 2 & \omega 2+1 & \omega 2+2 & \omega 2+3 & \omega 2+4 & \omega 2+5 & \dots \\
\omega 3 & \omega 3+1 & \omega 3+2 & \omega 3+3 & \omega 3+4 & \omega 3+5 & \dots \\
\omega 4 & \omega 4+1 & \omega 4+2 & \omega 4+3 & \omega 4+4 & \omega 4+5 & \dots \\
\omega 5 & \omega 5+1 & \omega 5+2 & \omega 5+3 & \omega 5+4 & \omega 5+5 & \dots \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots &
\end{array}$$

(Pour passer d'un ordinal à celui écrit à sa droite, on prend le successeur, alors que le premier ordinal de chaque ligne est obtenu comme la borne supérieure de tous les ordinaux précédents).

Regardons alors la première colonne. L'ensemble $\{0, \omega, \omega 2, \dots, \omega n, \dots\}$ n'a pas encore de borne supérieure. Il faut donc la rajouter obtenant ainsi l'ordinal ω^2. Vient ensuite ω^2+1,

[1] Comme le fait remarquer Moubinool, ω est simplement le symbole ∞ auquel on a coupé la tête.

[2] Que $\alpha+1$ ait un sens ou pas importe peu, disons pour l'instant qu'il s'agit d'une notation. C'est l'indice qui correspond à l'élément de E successeur de x_α.

etc. Rapidement il va arriver $\omega^2+\omega$ dont on imagine bien la définition puis $\omega^2+\omega 2$, $\omega^2+\omega 3$ et $\omega^2 2 = \omega^2+\omega^2$. Viendra un jour ω^3, ω^4, puis ω^n pour tout entier n. La borne supérieure de tout cela est alors naturellement ω^ω. En brûlant des étapes, on voit bien apparaître au loin ω^{ω^ω}, $\omega^{\omega^{\omega^\omega}}$ et puis aussi une tour d'exposants de hauteur ω : c'est l'ordinal ε_0. On peut encore aller plus loin. Pour l'instant, nous ne donnons pas de définition précise, mais il y aura ε_1, ε_2, ε_ω, $\varepsilon_{\varepsilon_0}$, $\varepsilon_{\varepsilon_{\varepsilon_0}}$ et puis une tour d'indices de hauteur ω qui correspond à l'ordinal ζ_0. On peut refaire le même tour avec les ζ : l'ordinal $\zeta_{\zeta_{\zeta_{\cdot^{\cdot^{\cdot}}}}}$ est celui que l'on note η_0.

2 Les ordinaux récursifs

2.1 Bons ordres pour les premiers ordinaux décrits

De façon générale, si E est un ensemble bien ordonné, il est possible de « numéroter » ses éléments par les « premiers » ordinaux. Le plus petit ordinal qui ne correspond ainsi à aucun élément de E est ce que l'on appelle le *type d'ordre* de E. Par exemple si E est l'ensemble des entiers naturels que l'on munit de l'ordre standard, on numérote les éléments de E par les entiers naturels qui sont les premiers ordinaux, et le plus petit ordinal qui n'est pas utilisé est ω ; c'est donc lui le type d'ordre de $\mathbb{N}$.

Maintenant, si on se donne un ordinal α, il est possible de construire un ensemble bien ordonné E dont le type d'ordre est précisément α : il suffit de prendre l'ensemble des ordinaux plus petits que α que l'on ordonne de la façon naturelle. Pour beaucoup d'ordinaux donnés précédemment, il n'est pas très difficile de décrire de façon plus explicite un ensemble et un ordre qui conviennent. Par exemple, on a déjà vu que l'ensemble des entiers naturels ordonné de la façon classique avait pour type d'ordre ω. Pour construire un ordre de type $\omega+1$, il suffit de rajouter un élément au bout que l'on peut naturellement noter ∞. Ainsi $\mathbb{N}\sqcup\{\infty\}$ ordonné de la façon à laquelle on pense (*i.e.* ∞ est plus grand que n'importe quel entier, entiers qui sont ordonnés de la façon classique) définit un ordre de type $\omega+1$.

L'ordinal $\omega 2$ correspond, quant à lui, à deux copies des entiers mises une à la suite de l'autre. Plus précisément, considérons le produit cartésien $\{0,1\}\times\mathbb{N}$ et munissons-le de l'ordre lexicographique[3]. Dans notre cas, cela veut simplement dire que l'on décrète que les éléments de la forme $(0,n)$ sont tous plus petits que ceux de la forme $(1,m)$ et que les couples (ε,n) à (ε,m) sont rangés dans le même ordre que n et m (ε valant bien sûr 0 ou 1).

Plus généralement, un bon ordre de type ωn est donné par l'ordre lexicographique sur le produit cartésien $\{0,\dots,n-1\}\times\mathbb{N}$. L'ordinal ω^2 correspond à l'ordre lexicographique sur $\mathbb{N}^2$. Si on regarde comment est obtenu l'ordinal $\omega^2+\omega$, on se rend compte qu'un ordre qui le décrit est l'union disjointe[4] $\mathbb{N}^2\sqcup\mathbb{N}$ sur laquelle on décrète que les éléments de $\mathbb{N}^2$ (ordonnés entre eux comme ils le savent) sont plus petits que ceux de $\mathbb{N}$ (ordonnés entre eux comme ils le savent également).

[3] C'est l'ordre du dictionnaire sur un produit cartésien. Rigoureusement et de façon générale, si $E_1,\dots,E_n$ sont des ensembles ordonnés, on peut définir un ordre sur $E_1\times\ldots\times E_n$ de la façon suivante. On dit que $(x_1,\dots,x_n)<(y_1,\dots,y_n)$ si ces deux n-uplets sont distincts et si le plus petit indice i pour lequel $x_i\neq y_i$ est tel que $x_i<y_i$.

[4] Cela signifie simplement que l'on suppose que les ensembles que l'on réunit ne vont pas avoir des contributions communes. Ici, précisément, les éléments de $\mathbb{N}^2$ et ceux de $\mathbb{N}$ sont bien des entités distinctes, on ne fait donc aucune identification.

Ensuite, ω^3 est représenté par l'ordre lexicographique sur $\mathbb{N}^3$. De même ω^n correspond à l'ordre lexicographique sur $\mathbb{N}^n$. Trouver un bon ordre de type ω^ω est un peu plus délicat, mais pas encore insurmontable. La définition que l'on a donnée de ω^ω suggère de prendre la réunion des $\mathbb{N}^n$, n variant dans les entiers naturels, en prenant soin d'identifier un n-uplet $(x_1, \ldots, x_n)$ aux m-uplets (avec $m \geqslant n$) $(0, \ldots, 0, x_1, \ldots, x_n)$. Une façon un peu plus simple de décrire cet ensemble consiste à considérer l'ensemble des suites d'entiers nulles à partir d'un certain rang. L'identification est presque immédiate : à un n-uplet $(x_1, \ldots, x_n)$ on fait correspondre la suite qui commence par $x_n, \ldots, x_1$ et qui termine avec des 0.

Il reste à définir l'ordre sur cet ensemble. Pour cela, on se rappelle que les ordres que l'on avait sur les $\mathbb{N}^n$ étaient les ordres lexicographiques. Muni de ces bases, avec suffisamment d'intelligence et patience, on déduit que l'ordre sur les suites est décrit de la façon suivante. Pour comparer les suites (x_n) et (y_n), on regarde le plus grand indice i tel que $x_i \neq y_i$ (un tel indice existe forcément car les suites (x_n) et (y_n) sont supposées nulles et donc égales à partir d'un certain rang). L'ordre entre les entiers x_i et y_i définit alors l'ordre sur les suites. Le lecteur pourra s'amuser à vérifier assidûment tout ce qui précède et à écrire les premières suites (et les suivantes) pour l'ordre défini.

Le tableau suivant récapitule les résultats que l'on vient de citer :

Ordinal	**Bon ordre**
ω	$\mathbb{N}$
$\omega + 1$	$\mathbb{N} \sqcup \{\infty\}$
ωn	$\{0, \ldots, n-1\} \times \mathbb{N}$
ω^2	$\mathbb{N}^2$
$\omega^2 + \omega$	$\mathbb{N}^2 \sqcup \mathbb{N}$
ω^n	$\mathbb{N}^n$
ω^ω	Suites d'entiers nulles à partir d'un certain rang

2.2 Objectif et définition

Notre but sera par la suite la description d'ensembles ordonnés de plus en plus compliqués pour atteindre des ordinaux de plus en plus vertigineux. On remarque en fait que plutôt que de permettre de complexifier à la fois les ensembles et les ordres, on peut imposer que l'ensemble sous-jacent soit toujours $\mathbb{N}$. En effet, les ensembles qui sont apparus précédemment sont tous en bijection avec $\mathbb{N}$ de façon plus ou moins simple certes mais toujours concrète.

On va donc s'intéresser à décrire des bons ordres sur $\mathbb{N}$ ayant pour type d'ordre des ordinaux de plus en plus grands. Par exemple, est-il possible d'atteindre ε_0, ζ_0 ou encore η_0 ? Cependant, avant de répondre à cette question pour le moins obscure, il est nécessaire d'expliquer ce que l'on entend ou peut entendre par « décrire ».

Le mot « décrire » dans ce qui précède est pratiquement un synonyme de « calculer ». Précisément, on dit qu'un ordre $\succ$ sur $\mathbb{N}$ est *calculable* s'il existe un algorithme (c'est-à-dire un programme informatique) qui prend en entrée deux entiers x et y et qui répond *plus petit* si le premier d'entre eux est plus petit (*i.e.* si $x \succ y$) et *plus grand* dans le cas contraire. Un ordinal qui est le type d'ordre d'un bon ordre sur $\mathbb{N}$ est dit *récursif*. La question qui nous motive est de déterminer les ordinaux récursifs.

Quelques remarques simples peuvent d'ores et déjà être faites. Il existe un nombre indénombrable d'ordinaux correspondant à des bons ordres sur $\mathbb{N}$. En effet, si ce nombre était dénombrable, on obtiendrait de fait une façon d'indicer les entiers avec *tous* ces ordinaux. Mais quel serait alors le type de ce bon ordre ? D'autre part, l'ensemble des programmes informatiques est dénombrable : tout programme informatique est une suite finie de caractères ou de bits si l'on préfère. En conséquence, tous les ordinaux ne sont pas récursifs. Quel est donc le plus petit ordinal non récursif ? Peut-on en donner une description autre que « c'est le plus petit ordinal non récursif » ? Le sens de « décrire » est nécessairement différent cette fois-ci puisque par définition, on ne peut écrire un programme qui calcule un bon ordre sur $\mathbb{N}$ dont le type d'ordre est cet ordinal. Cependant, il est peut-être possible d'en dire autre chose.

2.3 Un premier méta-programme : `successeur`

Pour essayer d'atteindre l'objectif que nous nous sommes fixés, nous allons écrire des *méta-programmes*, c'est-à-dire des programmes manipulant d'autres programmes. Exactement, pour nous, un méta-programme est un programme qui prend en argument des programmes $P_1, \ldots, P_n$ calculant des bons ordres sur $\mathbb{N}$ de type respectif $\alpha_1, \ldots, \alpha_n$, et qui calcule un bon ordre dont le type d'ordre est donné par une certaine fonction des α_i. L'idée est bien entendu d'utiliser ces méta-programmes pour atteindre un grand nombre d'ordinaux « d'un seul coup » et par le fait de brûler des étapes dans l'ascension de l'échelle des ordinaux.

Donnons tout de suite un premier exemple simple : la fonction `successeur`. Par définition, cette fonction prend en argument un programme `P` qui calcule un bon ordre de type α sur $\mathbb{N}$ et calcule un bon ordre de type $\alpha+1$. Voyons comment on écrit cela proprement dans un langage de programmation. D'abord il s'agit de se demander quelles est la nature (ou les *types*) des arguments de la fonction. Faisons en premier lieu l'analyse sur le programme `P`. Par définition celui-ci calcule un ordre sur $\mathbb{N}$, il prend donc en argument deux entiers et retourne l'un des mots clés *plus petit*, *plus grand* ou *égal*. En langage `C`, on résume cela par la syntaxe suivante appelée *prototype* :

```
ordre P (int n, int m);
```

Précisément la ligne qui précède signifie que le programme `P` prend en argument deux entrées appelées respectivement `n` et `m`, toutes deux de type `int` (qui signifie *entier*) et retourne une sortie de type `ordre` (qui ne peut prendre que les trois valeurs *plus petit*, *plus grand* ou *égal*). Le mot clé `int` est spécifique au langage `C`, il serait donc mal vu d'en utiliser un autre. Par contre, le type `ordre` est un type que l'on a défini nous-mêmes

Voyons à présent quel est le prototype de la fonction `successeur`. Celle-ci prend en entrée en premier lieu le programme `P`. Il nous faut donc donner un type au programme lui-même. Comme ce programme implémente une fonction qui associe un objet de type `ordre` à deux entiers, on convient que le type de `P` est `(int x int) -> ordre`, que l'on abrège par le mot clé `Ordinal`. En outre, le but de cette fonction est de calculer un ordre, elle doit donc prendre en arguments encore deux entiers et renvoyer un objet de type `ordre`. Finalement le prototype de la fonction `successeur` est :

```
ordre successeur (Ordinal P, int n, int m);
```

Le cahier des charges de la fonction `successeur` stipule que si `P` est un programme qui calcule un bon ordre de type α, alors le programme `successeur (P, _, _)` doit calculer

un bon ordre de type $\alpha+1$ où, lorsque P est fixé, le programme `successeur (P, _, _)` est celui qui prend en entrée deux entiers `n` et `m` et renvoie la valeur de retour de `successeur (P, n, m)`.

Écrivons enfin le programme `successeur` :

```
ordre successeur (Ordinal P, int n, int m) {
  if (n == 0 && m == 0) return EGAL;
  if (n == 0) return PLUS_GRAND;
  if (m == 0) return PLUS_PETIT;
  return P(n-1, m-1);
}
```

Ce programme, *via* le bon contexte, normalement compile. Précisons tout de suite que c'est un souci qui nous importera peu par la suite, le but étant plutôt de décrire des algorithmes sans souci d'efficacité que d'implémenter quoi que ce soit. Il faut peut-être préciser que « `==` » correspond au test de l'égalité et que « `&&` » correspond au *et* logique.

Voyons, chose plus intéressante, comment fonctionne ce programme : lorsque l'on a à comparer les entiers n et m, on teste d'abord (c'est le rôle des trois premières lignes) si l'un d'eux est nul, auquel cas on décrète que c'est lui le plus grand. Sinon, on compare *via* le programme `P` les entiers $n-1$ et $m-1$. En résumé, on commence par classer les entiers non nuls par le programme `P` et on rajoute au bout l'élément 0. Ainsi dit, il est clair que si `P` calcule l'ordinal α, le programme `successeur (P, _, _)` calcule un ordre qui, restreint aux entiers non nuls, est un bon ordre de type α. Ainsi puisque `successeur (P, _, _)` décrète que 0 est plus grand que tous les entiers, il calcule bien un bon ordre de type $\alpha+1$.

Ce qui précède assure que si α est récursif alors il en est de même de $\alpha+1$. Le plus petit ordinal non récursif ne peut donc pas être *successeur* (*i.e.* de la forme $\beta+1$ pour un certain ordinal β). Ce n'est certes pas encore très impressionnant, mais nous allons écrire d'autres méta-programmes pour monter plus vite dans l'échelle des ordinaux.

2.4 Somme et produit d'ordinaux

Il est possible de définir des opérations sur les ordinaux, opérations qui étendent celles définies sur les entiers. Le but de ce paragraphe est de donner ces définitions et de montrer que si α et β sont deux ordinaux récursifs, il en est de même de leur somme et de leur produit.

Commençons par définir la somme. On considère α et β deux ordinaux (pas forcément récursifs) et A (resp. B) un ensemble bien ordonné de type α (resp. β). Soit $C = A \sqcup B$ l'union disjointe de A et de B. Si on n'aime pas parler d'union disjointe, on peut voir les éléments de C comme des couples, soit de la forme $(0, a)$ avec $a \in A$, soit de la forme $(1, b)$ avec $b \in B$. On définit alors un ordre sur C qui correspond à ce que l'on attend de l'ordre lexicographique : plus précisément, on decrète que les éléments de la forme $(0, a)$ sont tous inférieurs à ceux de la forme $(1, b)$, et que (ε, x) et (ε, x') sont rangés dans le même ordre que x et x'. On vérifie C devient ainsi un ensemble bien ordonné dont on note γ le type. C'est par définition la *somme* de α et de β. On note $\gamma = \alpha + \beta$.

Voyons des exemples. Prenons pour commencer $\alpha = \omega$ et $\beta = 1$. Un ensemble A convenable est alors $\mathbb{N}$. Tout singleton convient pour B : prenons au hasard $B = \{\infty\}$.

L'ensemble $C = A \sqcup B$ est alors $\mathbb{N} \sqcup \{\infty\}$ sur lequel l'ordre est transparent. L'ordinal correspondant est donc celui que l'on a appelé $\omega + 1$. En bref, $\omega + 1 = \omega + 1$.

Si par contre, on prend $\alpha = 1$ et $\beta = \omega$, les choses se passent différemment : dans l'ordre induit sur C, l'élément ∞ est plus petit que tous les nombres, appelons-le donc plutôt -1. Ainsi, C est l'ensemble des entiers plus grands ou égaux à -1. Mais alors, quitte à remplacer chaque entier par son suivant, opération qui ne modifie pas l'ordre, on voit que C n'est autre que l'ensemble des entiers. On arrive finalement à $1 + \omega = \omega$, et non pas $\omega + 1$. Attention donc : l'addition n'est *pas* commutative.

Voyons maintenant la multiplication. Comme précédemment, on considère α et β deux ordinaux, puis A (resp. B) un bon ordre de type α (resp. β). L'ensemble C est défini comme le produit cartésien $C = B \times A$ que l'on munit de l'ordre lexicographique. Cet ordre est un bon ordre, et de fait correspond à un certain ordinal γ appelé *produit* de α et de β et noté $\gamma = \alpha \cdot \beta$.

Comme précédemment, examinons les premiers exemples. Évidemment, les choix $\alpha = 1$ ou $\beta = 1$, ne sont pas très intéressants puisqu'il est facile de voir que $1 \cdot \alpha = \alpha \cdot 1 = \alpha$ pour tout ordinal α. Considérons donc $\alpha = \omega$ et $\beta = 2$. Un ensemble A possible est encore $\mathbb{N}$. Un ensemble B possible est le doubleton $\{0, 1\}$ où on s'accorde à dire que $0 < 1$. Le produit cartésien est alors $C = \{0, 1\} \times \mathbb{N}$. Il correspond à l'ordinal $\omega 2$. Ainsi $\omega \cdot 2 = \omega 2$, c'est rassurant !

De même que pour l'addition, cela ne marche plus si l'on prend $\alpha = 2$ et $\beta = \omega$. Le produit cartésien est cette fois-ci $\mathbb{N} \times \{0, 1\}$. La liste des premiers éléments est :

$$(0,0), (0,1), (1,0), (1,1), (2,0), (2,1), (3,0), \ldots, (n-1,1), (n,0), (n,1), (n+1,0), \ldots$$

Cet ordre ressemble encore fort à ω. Ainsi $2 \cdot \omega = \omega$. La multiplication, non plus, n'est *pas* commutative. Tant pis encore une fois, il ne faut pas en faire un drame.

On dispose toutefois de certaines propriétés comme :
Pour l'addition :
- (élément neutre) $\alpha + 0 = 0 + \alpha = \alpha$
- (associativité) $\alpha + (\beta + \gamma) = (\alpha + \beta) + \gamma$

Pour la multiplication :
- (élément neutre) $\alpha \cdot 1 = 1 \cdot \alpha = \alpha$
- (associativité) $\alpha \cdot (\beta \cdot \gamma) = (\alpha \cdot \beta) \cdot \gamma$
- (élément absorbant) $\alpha \cdot 0 = 0 \cdot \alpha = 0$
- (distributivité) $\alpha \cdot (\beta + \gamma) = \alpha \cdot \beta + \alpha \cdot \gamma$

Il doit être possible d'écrire d'autres formules valables sur l'arithmétique des ordinaux mais il faut toutefois être très prudent. On attire par exemple l'attention du lecteur sur le fait que la distributivité n'est vraie que d'un côté (celui écrit ci-dessus), pas des deux.

La description précédente donne directement un méta-programme pour calculer la somme et un pour calculer le produit. Ces programmes peuvent s'écrire :

```
ordre somme (Ordinal P, Ordinal Q, int n, int m) {
  if (n est pair && m est pair) return P(n/2, m/2);
  if (n est impair && m est impair) return Q((n-1)/2, (m-1)/2);
  if (n est pair) return PLUS_PETIT;
  return PLUS_GRAND;
}
```

```
ordre produit (Ordinal P, Ordinal Q, int n, int m) {
  (n1, n2) = int_to_couple (n);
  (m1, m2) = int_to_couple (m);
  if (n1 == m1) return P(n2, m2);
  return Q(n1, m1);
}
```

Cette fois-ci, ces routines ne vont certainement pas compiler directement, surtout pour le produit, mais ce n'est pas le point crucial. Il faut préciser tout d'abord que `P` et `Q` sont censés être des programmes qui calculent respectivement les ordinaux α et β. Dans ce cas, le programme `somme (P, Q, _, _)` calcule un bon ordre de type $\alpha + \beta$ et le programme `produit (P, Q, _, _)` calcule un bon ordre de type $\alpha \cdot \beta$.

Nous laissons au lecteur le soin de comprendre le fonctionnement de ces routines. Signalons pour lui simplifier la tâche que la fonction `int_to_couple` prend en argument un entier et renvoie un couple d'entiers, établissant ainsi une bijection entre $\mathbb{N}$ et $\mathbb{N}^2$. Il existe des formules explicites et relativement simples pour cela... ce n'est toutefois pas l'objet de les détailler ici.

2.5 Moins de contraintes sur les programmes !

On se rend compte que les programmes précédents, ceux de type `Ordinal` se doivent de calculer des bons ordres sur $\mathbb{N}$ tout entier. En particulier, il n'est pas possible pour un tel programme de représenter un ordinal *fini* (*i.e.* strictement inférieur à ω, ou si l'on préfère simplement un entier). Les méta-programmes précédents ne peuvent donc s'appliquer qu'à des ordinaux infinis, ce qui n'est en soi pas dramatique, nous l'accordons.

On préfère toutefois relâcher quelque peu les contraintes que l'on a fixées pour les programmes de type `Ordinal`. Déjà, admettons que le bon ordre que calcule un tel programme n'est pas forcément défini sur tous les entiers mais seulement sur une partie $A \subset \mathbb{N}$. Dans la pratique, on ajoute une quatrième possibilité pour le type `ordre` nommée *erreur* et lorsque le programme `P` de type `Ordinal` est appelé avec les valeurs `n` et `m` et que soit n, soit m n'appartient pas à A il doit retourner *erreur*. Ainsi, on peut maintenant écrire un programme qui calcule l'ordinal fini 32 :

```
ordre trente-deux (int n, int m) {
  if (n >= 32 || m >= 32) return ERREUR;
  if (n < m) return PLUS_PETIT;
  if (n > m) return PLUS_GRAND;
  return EGAL;
}
```

(On voit ici apparaître le symbole « `||` » qui représente en `C` le *ou* logique). Dans ce cas, il est bien vrai que le programme `somme (trente-deux, trente-deux, _, _)` calcule l'ordinal fini 64.

Autorisons encore une deuxième légèreté qui va nous être utile par la suite : on ne demande plus à `P` de calculer un bon ordre, mais plutôt un bon pré-ordre. Autrement dit, on autorise à deux entiers distincts la possibilité de représenter le même élément. En particulier, un tel programme `P` peut très bien répondre *égal* lorsqu'on lui donne les nombres 17 et 42 en argument.

La procédure *somme* donnée précédemment s'adapte à cette nouvelle situation sans aucune modification. La procédure `produit` par contre fait à un moment un test d'égalité, qui va devoir être remplacé par un test d'« égalité au sens de `Q` ». Au final, il faut écrire :

```
ordre produit (Ordinal P, Ordinal Q, int n, int m) {
  (n1, n2) = int_to_couple (n);
  (m1, m2) = int_to_couple (m);
  if (Q(n1, m1) == EGAL) return P(n2, m2);
  return Q(n1, m1);
}
```

Bien sûr, on n'est pas obligé de faire le calcul de `Q(n1, m1)` deux fois, il aurait été plus intelligent de stocker le résultat.

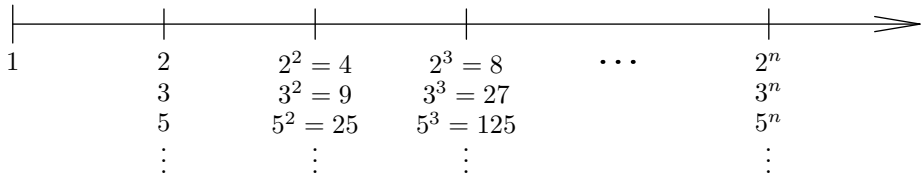
Finalement, il est important de signaler que l'on dispose d'un méta-programme prenant en argument un programme qui calcule un bon pré-ordre de type α avec possibilité de renvoyer *erreur* et qui calcule, lui, un bon ordre de type α, bon ordre défini sur $\mathbb{N}$ tout entier. Bien sûr, cela ne peut fonctionner si α est un ordinal fini, le fait est qu'il n'existe pas de bon ordre défini sur tout $\mathbb{N}$ dont le type soit un ordinal fini, $\mathbb{N}$ étant *malheureusement* infini. Ce méta-programme n'a rien de subtil ni de sorcier. Le voici :

```
ordre bonordre (Ordinal P, int n, int m) {
  i = 0; j = 0;
  while (i <= n && i <= m) {
    if (P(j, j) != ERREUR) {
      ok = 1;
      for (jj = 0; jj < j; jj++)
        if (P(jj, j) == EGAL) { ok = 0; break; }
      if (ok == 1) {
        if (i == n) np = j;
        if (i == m) mp = j;
        i = i + 1;
      }
      j = j + 1;
    }
  }
  return P(np, mp);
}
```

Le code précédent n'est probablement pas vraiment parlant, la syntaxe du `C` n'aidant pas. Comme le but n'est pas de faire un cours de `C`, nous n'allons pas nous attarder sur les joies de la syntaxe (en particulier celle du `for`). En retour, soyez gentils de ne pas faire remarquer l'oubli de déclaration des variables.

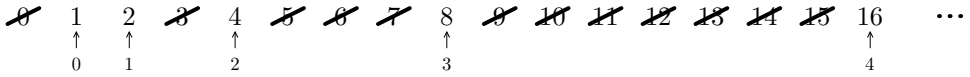
Le fonctionnement de cette routine est le suivant : pour chaque entier pris un par un, elle teste s'il fait partie de l'ensemble sur lequel le bon pré-ordre est défini (il s'agit de la condition `P(j, j) != ERREUR`), et l'élimine le cas échéant. Lorsque l'entier j est ainsi retenu, elle regarde si cet entier n'est pas une redondance d'un entier précédemment traité, auquel cas il est à nouveau éliminé. Finalement, elle dresse une liste des entiers non rejetés, et pour comparer n et m, elle compare *via* le programme `P` les entiers apparaissant en n-ième et m-ième position dans la liste.

Voici un exemple. Supposons que `P` calcule le bon pré-ordre donné par le diagramme ci-après :



Ce diagramme signifie que si `P` reçoit les entiers `n` et `m`, il commence par vérifier que chacune de ces deux valeurs puissent s'écrire sous la forme $n = p^{n'}$ et $m = q^{m'}$ où p et q sont des nombres premiers et n' et m' des entiers. Si ce n'est pas le cas, `P` renvoie *erreur* et si c'est le cas, `P` compare les exposants n' et m'. On obtient bien ainsi un bon pré-ordre de type ω.

Appliquer à ce programme `P` la routine `bonordre` revient dans un premier temps à lister tous les entiers et à en éliminer certains. Plus précisément, on a :



On commence par éliminer 0 qui ne fait pas partie des éléments classés par `P`. Ensuite, on garde 1 puisqu'il apparaît dans le classement et n'est *égal* (au sens de `P`) à aucun entier plus petit (au sens classique) que lui. De même, on garde 2, mais on élimine 3 car il est *égal* (au sens de `P`) à 2. On garde 4. On élimine 5 qui est *égal* à 2. On élimine 6 qui ne fait pas partie des entiers classés. On élimine 7 qui est encore *égal* à 2 mais par contre, on garde 8 et ainsi de suite. Il n'est pas difficile en fait de se convaincre que les entiers qui ne sont pas éliminés sont exactement les puissances de 2.

On numérote ensuite (à partir de 0) les entiers non barrés et pour comparer n et m, on compare au sens de `P` les entiers de cette liste numérotés n et m. On constate ici que l'on retrouve l'ordre usuel sur $\mathbb{N}$ qui est bien un bon ordre de type ω.

Remarquons que cette nouvelle description permet de prouver facilement que si α est un ordinal récursif et si $\beta < \alpha$, alors β est lui-même récursif. En effet, supposons que `P` soit un programme qui calcule un bon ordre sur $\mathbb{N}$ (tout entier) de type α. Puisque $\beta < \alpha$ il existe un entier M tel que le bon ordre induit par celui calculé par `P` sur la partie :

$$A = \{n \in \mathbb{N} \quad | \quad n < M \text{ pour l'ordre calculé par } \texttt{P}\}$$

soit de type β. Il s'agit donc de modifier le programme `P` pour qu'il renvoie *erreur* dans le cas où l'un des entiers passés est supérieur ou égal au sens de `P` à M. Si l'on préfère, on peut aussi écrire un nouveau programme :

```
ordre P_beta (int n, int m) {
  if (P(n, M) !=  PLUS_PETIT) return ERREUR;
  if (P(m, M) !=  PLUS_PETIT) return ERREUR;
  return P(n, m);
}
```

2.6 Puissance et forme normale de Cantor

Il est possible de définir l'exponentiation des ordinaux de manière analogue à ce qui a été fait pour l'addition et la multiplication : étant donnés un ensemble A bien ordonné de type α et un ensemble B bien ordonné de type β, on fait une construction qui aboutit à un nouvel ensemble bien ordonné C dont le type est par définition α^β. Cependant, nous allons procéder différemment.

On fixe un ordinal α et on définit par récurrence :

i) $\alpha^0 = 1$
ii) si $\beta = \gamma + 1$, $\alpha^\beta = \alpha^\gamma \cdot \alpha$
iii) sinon, $\alpha^\beta = \sup_{\gamma<\beta} \alpha^\gamma$

La définition précédente permet de donner un sens à α^β pour tout ordinal β, α restant toujours fixé. En effet, si ce n'était pas le cas, il y aurait un plus petit ordinal β pour lequel α^β ne serait pas défini, mais cela n'est pas possible puisque α^β est défini dans tous les cas en fonction des précédents.

Voyons ce que l'on obtient si $\alpha = 2$. Si $\beta = n$ est un entier, on voit facilement que $2^\beta = 2^n$, l'exponentiation dans le membre de droite étant celle que l'on est habitué à pratiquer. Maintenant par définition 2^ω est la borne supérieure des 2^n où n parcourt les entiers. Ainsi $2^\omega = \omega$. Ensuite $2^{\omega+1} = 2^\omega \cdot 2 = \omega 2$ et ainsi de suite $2^{\omega+n} = \omega 2^n$. De cette façon, on aboutit à $2^{\omega 2} = \omega^2$ puis à $2^{\omega n+m} = \omega^n 2^m$, n et m étant des entiers. De fait $2^{\omega^2} = \omega^\omega$, *etc.*

L'exponentiation des ordinaux vérifie au moins les propriétés suivantes :
- $\alpha^{\beta+\gamma} = \alpha^\beta \cdot \alpha^\gamma$
- $\alpha^{\beta\cdot\gamma} = \left(\alpha^\beta\right)^\gamma$

Il faut noter qu'il est tout à fait possible de définir également l'addition et la multiplication des ordinaux de la manière précédente. Plus précisément, on fixe toujours un ordinal α et on pose :
Pour l'addition
i) $\alpha + 0 = \alpha$
ii) si $\beta = \gamma + 1$, $\alpha + \beta = (\alpha + \gamma) + 1$
iii) sinon, $\alpha + \beta = \sup_{\gamma<\beta} \alpha + \gamma$
(Il faut noter que l'opération « $+1$ » se définit sans problème ; c'est simplement prendre le successeur).
Pour la multiplication
i) $\alpha \cdot 0 = 0$
ii) si $\beta = \gamma + 1$, $\alpha \cdot \beta = \alpha \cdot \gamma + \alpha$
iii) sinon, $\alpha \cdot \beta = \sup_{\gamma<\beta} \alpha \cdot \gamma$

Nous n'allons pas vérifier que ces définitions coïncident avec celles qui avaient été données auparavant. Ce n'est pas difficile et nous laissons cette tâche au lecteur.

Ayant défini l'exponentiation de deux ordinaux, on aimerait écrire un méta-programme qui, à partir d'un programme `P` qui calcule un bon ordre de type α, puisse calculer un bon ordre de type ω^α. Nous n'avons certainement pas besoin pour cela d'établir la propriété suivante mais elle permet probablement de mieux visualiser la situation :

Propriété 1. *Soit $\gamma < \omega^\alpha$. Alors il existe un* unique *entier n, une* unique *suite strictement décroissante (et donc finie*[5]*) d'ordinaux strictement inférieurs à α, $e_1 > e_2 > \ldots > e_n$, et une* unique *suite finie d'entiers $a_1, \ldots, a_n$, le tout tel que :*

$$\gamma = \omega^{e_1} a_1 + \ldots + \omega^{e_n} a_n$$

Ébauchons la démonstration de cette propriété. L'idée est de procéder par « récurrence » et donc d'écrire simplement :

$$\gamma = \omega^{e_1} a_1 + \delta$$

[5] Pourquoi au fait ?

où a_1 est un entier, $e_i < \alpha$ et δ est un reste strictement inférieur à ω^{e_1}. Si on parvient à cela, il suffira pour conclure, d'appliquer le même résultat au reste δ fournissant une nouvelle écriture :

$$\gamma = \omega^{e_1} a_1 + \omega^{e_2} a_2 + \delta$$

où δ est un nouveau reste strictement inférieur à ω^{e_2}. On continue ainsi de suite, pour construire une suite strictement décroissante d'ordinaux $e_1 > e_2 > \ldots > e_n$. Il s'agit de prouver que cette construction s'arrête forcément au bout d'un temps fini, c'est-à-dire que l'on tombe finalement sur un reste $\delta = 0$. Mais ceci est impliqué par le fait qu'il n'existe pas de suites strictement décroissantes d'ordinaux et infinies ; on ne pourra donc pas rajouter des exposants éternellement.

Il ne reste donc plus qu'à prouver la propriété énoncée ci-dessus. Pour cela considérons le plus petit ordinal β tel que $\gamma < \omega^\beta$. Cela signifie d'une part que $\gamma < \omega^\beta$ et d'autre part que pour tout $\beta' < \beta$, on a $\gamma \geqslant \omega^{\beta'}$. Montrons que β s'écrit $\beta = e_1 + 1$ pour un certain ordinal e_1. On raisonne par l'absurde : si ce n'était pas le cas, par définition de l'exponentiation, on aurait :

$$\omega^\beta = \sup_{\beta' < \beta} \omega^{\beta'}$$

Mais tous les ordinaux dont on prend la borne supérieure dans l'égalité précédente sont inférieurs ou égaux à γ ; il est en donc de même de cette borne supérieure. Or ω^β est lui strictement supérieur à γ. Voici notre contradiction.

De manière similaire, on prouve que le plus petit ordinal β tel que $\gamma < \omega^{e_1}\beta$ est de la forme $\beta = a_1 + 1$. On remarque ensuite que ω vérifie $\gamma < \omega^{e_1} \cdot \omega = \omega^{e_1+1}$ et donc que forcément $\beta \leqslant \omega$ puis que $a_1 < \omega$. L'ordinal a_1 est donc un entier. On a alors l'inégalité suivante :

$$\omega^{e_1} a_1 \leqslant \gamma < \omega^{e_1} (a_1 + 1) = \omega^{e_1} a_1 + \omega^{e_1}$$

qui permet de conclure quant à l'existence du reste δ ayant les propriétés voulues.

Ceci achève la démonstration.

Remarquons que cette écriture correspond à une décomposition en base ω de l'ordinal γ. En réalité, ceci reste valable de façon plus générale en remplaçant ω par un ordinal quelconque $\beta \geqslant 2$, les a_i étant alors des ordinaux strictement inférieurs à β comme dans le cas classique.

Il reste à préciser que la décomposition en base ω permet de comparer facilement deux ordinaux : il suffit pour cela de comparer les a_i exactement de la même façon que l'on compare les chiffres de deux entiers écrits dans la même base pour comparer ces entiers.

Donnons une autre formulation de la propriété précédente :

Propriété 2 (Forme normale de Cantor)**.** *Soit $\gamma < \omega^\alpha$. Alors il existe un* unique *entier n, une* unique *suite décroissante (au sens large) d'ordinaux strictement inférieurs à α, $e_1 \geqslant e_2 \geqslant \ldots \geqslant e_n$ tel que :*

$$\gamma = \omega^{e_1} + \ldots + \omega^{e_n}$$

Pour prouver ce corollaire, il suffit d'appliquer la propriété précédente et de remarquer que $\omega^e a = \omega^e + \ldots + \omega^e$ (a fois).

Donnons-nous maintenant γ et γ' deux ordinaux qui s'écrivent respectivement :

$$\begin{aligned} \gamma &= \omega^{e_1} + \ldots + \omega^{e_n} \\ \gamma' &= \omega^{e'_1} + \ldots + \omega^{e'_m} \end{aligned}$$

n et m étant des entiers, $e_1 \geqslant e_2 \geqslant \ldots \geqslant e_n$ et $e'_1 \geqslant e'_2 \geqslant \ldots \geqslant e'_m$. Comment peut-on comparer γ et γ' sur leur décomposition en base ω ? On regarde simplement les premiers exposants e_1 et e'_1 : si $e_1 < e'_1$, alors $\gamma < \gamma'$; si $e_1 > e'_1$, alors $\gamma > \gamma'$; si $e_1 = e'_1$, on compare de la même façon les seconds exposants e_2 et e'_2 et ainsi de suite. Bien entendu, si l'on arrive à court d'exposant pour un des deux ordinaux et qu'il en reste encore dans l'autre, l'ordinal avec des exposants excédentaires est le plus grand.

Bien que nous ne l'écrirons pas précisément, les remarques précédentes peuvent facilement se transformer en un programme de prototype

```
ordre puissances_de_omega (Ordinal P, int n, int m);
```

qui, à partir d'un programme `P` calculant un bon ordre de type α, calcule un bon ordre de type ω^α. L'idée est très simple : les ordinaux strictement inférieurs à ω^α forment un ensemble naturellement bien ordonné de type d'ordre ω^α, et on peut représenter ces ordinaux par des entiers.

Plus exactement, le programme commence par convertir n et m en une suite finie d'entiers, puis il vérifie que ces suites sont décroissantes au sens large et finalement il applique alors la méthode de comparaison décrite précédemment. Schématiquement, on peut écrire :

```
ordre puissances_de_omega (Ordinal P, int n, int m) {
  // On convertit n et m en des suites
  N = suite (n); M = suite (m);

  // On teste si N et M sont bien décroissantes
  for (i = 1; i < longueur (N); i++) {
    r = P(N[i-1], N[i]);
    if (r == PLUS_PETIT || r == ERREUR) return ERREUR;
  }
  for (i = 1; i < longueur (M); i++) {
    r = P(M[i-1], M[i]);
    if (r == PLUS_PETIT || r == ERREUR) return ERREUR;
  }

  // On compare
  for (i = 0; i < longueur (N) && i < longueur (M); i++) {
    r = P(N[i], M[i]);
    if (r != EGAL) return r;
  }
  if (longueur (N) < longueur (M)) return PLUS_PETIT;
  if (longueur (N) > longueur (M)) return PLUS_GRAND;
  return EGAL;
}
```

tout en laissant au lecteur le soin d'écrire un code lisible et efficace. Signalons que dans le programme `puissances_de_omega`, la procédure externe `suite` doit avoir le comportement suivant : elle prend en argument un entier et renvoie une suite finie d'entiers, établissant ainsi une bijection entre ces deux ensembles. La procédure `longueur`, quant à elle, calcule la longueur d'une suite finie d'entiers.

2.7 Jusqu'à ε_0 et au-delà

Voyons ce que donne la description précédente lorsque $\alpha = \varepsilon_0$. On rappelle que ε_0 est défini comme la limite de la suite croissante $1, \omega, \omega^\omega, \omega^{\omega^\omega}, \ldots$. Il s'agit maintenant de calculer ω^{ε_0} ; par définition on a :

$$\omega^{\varepsilon_0} = \sup_{\beta<\varepsilon_0} \omega^\beta$$

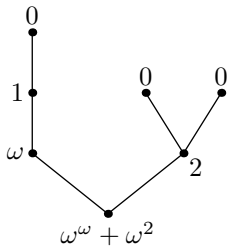
Pour la dernière borne supérieure, on peut en fait se restreindre aux β qui s'écrivent comme une tour de puissances d'ω. Ceci démontre que finalement $\omega^{\varepsilon_0} = \varepsilon_0$.

Dans ces conditions, la propriété 2 se réécrit sous la forme :

Propriété 3. *Soit $\gamma < \varepsilon_0$. Alors il existe un* unique *entier n, une* unique *suite décroissante (au sens large) d'ordinaux strictement inférieurs à ε_0, $e_1 \geqslant e_2 \geqslant \ldots \geqslant e_n$ tel que :*

$$\gamma = \omega^{e_1} + \ldots + \omega^{e_n}$$

Ainsi, un ordinal inférieur à ε_0 n'est autre qu'une suite décroissante finie d'ordinaux inférieurs à ε_0. On peut représenter cette correspondance sous forme d'arbre : l'arbre qui n'a qu'une racine représente l'ordinal 0 et l'arbre formé d'une racine et de n sous-arbres correspondant respectivement aux ordinaux $e_1, \ldots, e_n$ supposés triés par ordre décroissant, représente l'ordinal $\omega^{e_1} + \ldots + \omega^{e_n}$. Par exemple, l'ordinal $\omega^\omega + \omega^2$ est représenté par :



Sur le dessin précédent, à côté de chaque nœud est reporté l'ordinal qui correspond au sous-arbre dont le nœud en question est la racine. On vérifie sans mal que l'ordinal associé à un nœud se calcule bien en exécutant l'opération décrite précédemment avec les ordinaux associés à chacun des fils du nœud en question.

Réciproquement, on comprend bien comment construire l'arbre associé à un ordinal $\gamma < \varepsilon_0$ donné. On écrit γ sous sa forme normale de Cantor et on dessine l'arbre correspondant à chacun des exposants qui apparaissent. On rajoute finalement un point tout en bas, et on le relie à la racine de chacun des arbres tracés. Ce point devient la nouvelle racine.

Toutefois, pour que cette construction fonctionne, il faut s'assurer que l'on ne dessine pas des arbres indéfiniment. Il suffit pour cela de remarquer que si $\gamma < \varepsilon_0$ s'écrit $\gamma = \omega^{e_1} + \ldots + \omega^{e_n}$, chacun des exposants e_i est strictement inférieur à γ. Pour cela, il suffit de prouver que $\gamma < \omega^\gamma$. Mais, si ce n'était pas le cas, on aurait $\gamma \geqslant \omega^\gamma$. D'autre part, on a $\gamma \geqslant 0$ et donc $\omega^\gamma \geqslant \omega^0 = 1$. Cela prouve que $\gamma \geqslant 1$. On recommence : il vient $\omega^\gamma \geqslant \omega^1 = \omega$ et donc $\gamma \geqslant \omega$. Puis $\gamma \geqslant \omega^\omega$, puis $\gamma \geqslant \omega^{\omega^\omega}$ et ainsi de suite. Finalement, à la limite, on obtient $\gamma \geqslant \varepsilon_0$, ce qui est supposé faux.

On a désormais une représention utilisable des ordinaux inférieurs à ε_0 et on est donc en mesure d'écrire un programme qui calcule un bon ordre de type ε_0. Évidemment, il reste à comprendre comment on détermine, étant donnés deux arbres, lequel est le plus petit, mais nous l'avons déjà dit maintes fois dans d'autres contextes et nous laissons le lecteur

le retrouver. Pour finalement écrire ce programme, il nous faut disposer d'une fonction qui établisse une bijection entre les entiers et les arbres. Là encore, il n'est pas bien difficile de construire une telle chose et nous n'allons pas décrire les divers procédés et les comparer pour choisir le meilleur.

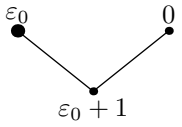
Donnons, à la surprise générale, une autre méthode plus subtile mais plus directe pour aboutir au résultat. Elle est basée sur l'égalité $\omega^{\varepsilon_0} = \varepsilon_0$. On a déjà écrit un méta-programme `puissances_de_omega` qui prend en argument un programme qui calcule un bon ordre de type α et qui en retour calcule un bon ordre de type ω^α. Si ce programme pouvait s'appeler lui-même, il calculerait probablement un bon ordre de type α vérifiant $\omega^\alpha = \alpha$ et donc probablement $\alpha = \varepsilon_0$. Et de fait, c'est le cas : même que nous ne le justifierons pas, le programme qui suit calcule un bon ordre de type ε_0.

```
ordre epsilon_zero (int n, int m) {
  return puissances_de_omega (epsilon_zero, n, m);
}
```

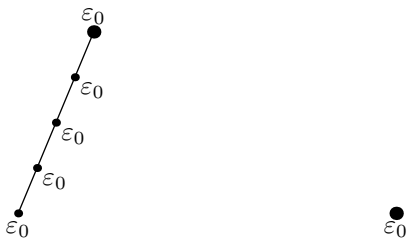
Évidemment tout cela ne doit plus vouloir dire grand chose en `C` (du moins écrit de cette façon) qui n'est pas un langage fonctionnel. Mais encore une fois, on ne va pas se traumatiser pour de telles broutilles.

Essayons désormais de dépasser ε_0. Le problème si l'on veut recopier ce qui précède est que l'écriture de ε_0 sous forme normale de Cantor est $\varepsilon_0 = \omega^{\varepsilon_0}$, qui fait intervenir à nouveau ε_0. On tourne en rond.

La solution consiste à rajouter une nouvelle entité : c'est un arbre, réduit à une simple racine, mais une racine de nature différente, une racine plus « grosse ». Cet arbre est celui qui symbolise ε_0. On n'a alors plus de problème pour écrire ε_0, ni même ε_0+1 qui correspond à :



puisque sa forme normale de Cantor est $\omega^{\varepsilon_0} + \omega^0$. Par un raisonnement analogue au précédent, on montre que cette méthode permet d'atteindre tous les ordinaux jusqu'à ε_1, deuxième plus petit ordinal α vérifiant $\omega^\alpha = \alpha$. Il y a toutefois un petit écueil qui n'est en fait pas un véritable problème : les deux arbres ci-dessous représentent le même ordinal, en l'occurrence ε_0.



Ce n'est en fait pas important puisque l'on a prévu dans le paragraphe 2.5 de se simplifier la vie en autorisant deux entiers distincts à représenter le même élément du bon ordre que l'on calcule. Il faut noter que cette simplification nous est bien utile ici.

Désormais que l'on a accepté un nouvel arbre pour ε_0 dans le but d'atteindre ε_1, on est bon pour en rajouter encore un nouveau qui correspond à ε_1. Ainsi, on peut décrire tous

les ordinaux inférieurs strictement à ce que l'on appelle ε_2, troisième ordinal α vérifiant $\omega^\alpha = \alpha$. En rajoutant ainsi n arbres, on obtient une description de tous les ordinaux inférieux à ε_n. Et en fait, avec les bonnes définitions qui vont venir dans le chapitre suivant, en rajoutant α arbres, on obtient une description de tous les ordinaux inférieurs à ε_α. Il est encore évidemment possible de coder tous ces nouveaux arbres par des entiers, et d'écrire directement un programme qui calcule un bon ordre de type ε_α dès que l'on dispose d'un programme qui calcule un bon ordre de type α ; il est en fait même possible d'écrire un méta-programme pour faire tout cela en famille.

Il est également possible d'écrire un programme qui calcule un bon ordre de type ε_1 et qui ne fait appel qu'à `puissances_de_omega`. Le voici :

```
ordre epsilon_un (int n, int m) {
  return puissances_de_omega (epsilon_un, n, m);
}
```

C'est un peu exagéré. Il faut dire en outre que la valeur ∞ (ou -1 si l'on préfère) est acceptée aussi bien pour l'entier `n` que pour l'entier `m` et que l'on modifie le programme `suite` (que l'on n'a de toute manière pas écrit) pour qu'il gère cette nouvelle valeur ∞ et pour que lorsqu'il est appelé avec la valeur ∞, il retourne la suite de longueur 1 égale à (∞). Bien sûr, cette nouvelle valeur ∞ est celle qui correspond à l'arbre de ε_0. Le fait que ∞ corresponde à (∞) est précisément l'identification des deux arbres :

3 Points fixes

3.1 Le plus petit point fixe

Il est en partie possible de refaire ce qui a été fait précédemment dans un contexte plus général : ε_0 est défini comme le plus petit ordinal α vérifiant $\omega^\alpha = \alpha$, ε_1 est le deuxième plus petit ordinal à vérifier cette condition, *etc.* Bref, les ε_i sont les points fixes successifs de la fonction $\alpha \mapsto \omega^\alpha$. Mais pourquoi se cantonner à cette fonction ? Que se passe-t-il si l'on en considère d'autres ?

Avant de faire quoi que ce soit, il faudrait délimiter un ensemble sur lequel seront définies ces fonctions. On serait tenté de considérer l'ensemble de tous les ordinaux, mais les théoriciens des ensembles nous certifient qu'un tel objet n'existe pas. On se cantonne donc modestement à l'ensemble[6] des ordinaux qui sont le type d'un bon ordre défini sur $\mathbb{N}$. Ces ordinaux sont qualifiés de *dénombrables* et leur ensemble est noté $\aleph_1$ (lire `aleph_1`).

On considère une fonction $f : \aleph_1 \to \aleph_1$ et on cherche des points fixes à f, c'est-à-dire des ordinaux α tel que $f(\alpha) = \alpha$. Pour trouver ces points fixes, dans le cas où f était la fonction définie par $f(\alpha) = \omega^\alpha$, on partait de 0, on calculait $f(0) = \omega^0 = 1$, puis $f(1) = \omega^1 = \omega$, puis $f(\omega) = \omega^\omega$, puis $f(\omega^\omega) = \omega^{\omega^\omega}$ et ainsi de suite. Autrement dit, on calculait les itérés successifs de f en 0, et on prenait la borne supérieure de tous les ordinaux obtenus. Elle convenait.

[6] Celui-ci existe bien : pour le construire, on peut quotienter l'ensemble des bons ordres définis sur une partie de $\mathbb{N}$ par la relation d'isomorphisme mettant en équivalence deux bons ordres s'il existe une bijection croissante de l'un dans l'autre.

Peut-on faire la même chose dans notre cas plus général ? On aimerait que la suite $0, f(0), f \circ f(0), \ldots, f \circ \ldots \circ f(0), \ldots$ soit croissante et que sa limite (la borne supérieure donc des ordinaux écrits) soit un point fixe de f. Bien sûr, cela ne risque pas d'être le cas de façon générale, on va avoir besoin d'hypothèses sur f. Dégageons-les.

Notons pour simplifier les écritures $f^{(n)}$ la composée $f \circ \ldots \circ f$ (n fois). On aimerait déjà que pour tout entier n, on ait $f^{(n+1)}(0) \geqslant f^{(n)}(0)$. Supposons plus fort : la fonction f est croissante (ce qui implique bien ce que l'on veut avec une récurrence facile). De plus, si on appelle λ la limite croissante (*i.e.* la borne supérieure) de la suite $\left(f^{(n)}(0)\right)$, on aimerait qu'elle vérifie $f(\lambda) = \lambda$ et cela découle d'une hypothèse de type continuité[7] : on suppose que si α_i est une famille dénombrable d'ordinaux, on a :

$$\sup_i f(\alpha_i) = f\left(\sup_i \alpha_i\right)$$

Il est ici opportum de remarquer que la borne supérieure d'un nombre dénombrable d'ordinaux de $\aleph_1$ est encore un ordinal de $\aleph_1$. En effet, on peut déjà supposer que la famille des α_i est indexée par $\mathbb{N}$ et, quitte à remplacer α_i par le plus grand des α_n pour $n \leqslant i$, que la suite (α_n) est croissante. Pour tout i, choisissons un ensemble bien ordonné dénombrable de type α_i et considérons la réunion de ces ensembles en faisant les identifications nécessaires pour obtenir ainsi un bon ordre de type α, borne supérieure de α_i. Cette réunion est une union dénombrable d'ensembles dénombrables et par le fait est dénombrable : α est donc bien un élément de $\aleph_1$.

Récapitulons finalement les hypothèses à faire sur la fonction f :
- (Croissance) pour tous ordinaux $\alpha \leqslant \beta$, $f(\alpha) \leqslant f(\beta)$
- (Continuité) pour toute famille dénombrable d'ordinaux (α_i), $\sup_i f(\alpha_i) = f(\sup_i \alpha_i)$

Sous ces hypothèses, comme nous venons de le voir, la suite $f^{(n)}(0)$ est croissante et sa limite λ est un point fixe de la fonction f. En fait, on a un petit complément : λ est le plus petit point fixe de la fonction f. En effet supposons que α soit un autre point fixe de f, c'est-à-dire qu'il vérifie $f(\alpha) = \alpha$. On a évidemment $\alpha \geqslant 0$. Par croissance de f, $\alpha = f(\alpha) \geqslant f(0)$, puis $\alpha = f(\alpha) \geqslant f^{(2)}(0)$. Ainsi, pour tout entier n, $\alpha \geqslant f^{(n)}(0)$ et donc $\alpha \geqslant \lambda$ en passant à la borne supérieure comme on le souhaitait.

Avant de continuer, présentons quelques exemples simples. Bien sûr, il y a la fonction $\alpha \mapsto \omega^\alpha$. Mais avant cela, il y a peut-être $f_1 : \alpha \mapsto 1+\alpha$ dont on peut vérifier (facilement *via* la définition de l'addition par « récurrence ») qu'elle vérifie les hypothèses voulues[8]. Lorsque l'on itère f_1 sur l'ordinal 0, on obtient successivement $0, 1, 2, \ldots$. La borne supérieure de ces nombres est par définition ω : c'est le plus petit point fixe de f_1. On vérifie bien par ailleurs que $f_1(\omega) = 1 + \omega = \omega$.

Traitons un second exemple, celui de la fonction $f_2 : \alpha \mapsto \omega + \alpha$. Elle est bien croissante et continue. La suite que l'on considère est $0, \omega, \omega 2, \ldots$. La borne supérieure et donc le plus petit point fixe de f_2 est ω^2. Considérons finalement $f_3 : \alpha \mapsto 1 + \omega \cdot \alpha$. (Le 1 ne sert à rien si ce n'est à éviter le canular selon lequel 0 serait point fixe de f_3). Ici, $f^{(n)}(0) = \omega^n$ et donc le plus petit point fixe de f_3 est ω^ω. Notez que de nombreux ordinaux peuvent

[7] Si vous avez déjà manipulé de la continuité, vous remarquerez immédiatement que la condition donnée juste en dessous correspond bien à la continuité habituelle pour les fonctions croissantes. En fait, si l'on munit $\aleph_1$ de la topologie induite par l'ordre naturel sur les ordinaux, la condition donnée correspond exactement à la continuité de l'application f.

[8] Remarquez que par contre, la fonction successeur $\alpha \mapsto \alpha + 1$ n'est *pas* continue, et de fait, elle n'admet pas de point fixe.

être définis comme étant les plus petits points fixes d'une fonction donnée et c'est souvent intéressant de pouvoir les voir ainsi.

Mais, quel est le rapport entre les points fixes et les ordinaux récursifs ? Précisément, si l'on suppose que la fonction f est calculable alors le plus petit point fixe de f est un ordinal récursif. Précisons tout de suite ce que l'on entend par le fait que la fonction f soit ou non *calculable*. Cela signifie qu'il existe un méta-programme qui prend en argument un programme qui calcule un bon ordre sur $\mathbb{N}$ de type α et qui en retour calcule un bon ordre sur $\mathbb{N}$ de type $f(\alpha)$. Le prototype d'un tel programme est donc, comme on en a l'habitude :

```
ordre f (Ordinal alpha, int n, int m);
```

Ainsi le type de `f` est `Ordinal x int x int -> ordre`, type qui n'avait pas encore été introduit. Comme nous allons en avoir besoin par la suite, désignons-le par le mot clé `Fonction`.

Notre objectif à présent est d'écrire un programme qui prend en argument un programme de type `Fonction` et qui calcule un bon ordre sur $\mathbb{N}$ de type λ, plus petit point fixe de f. Son prototype est donc :

```
ordre plus_petit_point_fixe (Fonction f, int n, int m);
```

Il s'agit d'un programme qui manipule des méta-programmes ; il peut à ce titre recevoir le nom de méta-méta-programme si le jeu nous amuse. Quoi qu'il en soit, un tel programme n'est pas difficile à écrire. Il suffit de recopier celui que l'on a donné pour ε_0 :

```
ordre plus_petit_point_fixe (Fonction f, int n, int m) {
  return f (plus_petit_point_fixe (f, _ _), n, m);
}
```

Ici, `plus_petit_point_fixe (f, _ _)` est le programme qui prend deux arguments `n` et `m` et qui retourne le résultat de `plus_petit_point_fixe (f, n, m)`. Cette fois-ci, de quelque façon qu'on s'y prenne, le `C` ne permet pas de telles constructions, mais cela n'a toujours pas une grande importance[9].

La procédure écrite ne fonctionne pas de façon générale : il faut en outre faire quelques hypothèses sur le programme `f`, typiquement que les appels récursifs qu'il fait à `alpha` ne fassent intervenir que des entiers strictement plus petits (au sens usuel) que les arguments `n` et `m` qui lui sont passés. Toutefois, on remarque qu'il ne s'agit pas d'hypothèses contraignantes, dans la pratique elles sont toujours satisfaites.

Illustrons le principe de ce programme avec la fonction $f_1 : \alpha \mapsto 1 + \alpha$. Implémentons pour cela dans un premier temps une fonction `f_un` :

```
ordre f_un (Ordinal alpha, int n, int m) {
  if (n == 0 && m == 0) return EGAL;
  if (n == 0) return PLUS_PETIT;
  if (m == 0) return PLUS_GRAND;
  return alpha (n-1, m-1);
}
```

[9] Un langage bien plus approprié serait sans aucun doute le caml mais j'ai un peu peur que sa syntaxe soit moins parlante au premier abord.

On remarque judicieusement que l'hypothèse dégagée précédemment est bien vérifiée par la routine `f_un` : le programme `alpha` est appelé avec $n-1$ qui est bien strictement inférieur à n et avec $m-1$ qui est bien strictement inférieur à m. Suivons à présent pas à pas l'exécution du programme `plus_petit_point_fixe (f_un, 7, 8)`. Pour simplifier les choses, donnons un nom au programme `plus_petit_point_fixe (f_un, _, _)`, disons `pppf_de_f_un`[10]. Il est décrit par le code suivant :

```
ordre pppf_de_f_un (int n, int m) {
  return f_un (pppf_de_f_un, n, m);
}
```

On remarque que les deux programmes `plus_petit_point_fixe (f_un, n, m)` et `pppf_de_f_un (n, m)` sont les mêmes, par définition. Ainsi, exécuter `plus_petit_point_fixe (f_un, 7, 8)` revient à exécuter `pppf_de_f_un (7, 8)` ou encore `f_un (pppf_de_f_un, 7, 8)`. Ni 7, ni 8 ne sont nuls, on passe donc à la troisième ligne du programme `f_un` et on est amené à exécuter `pppf_de_f_un (6, 7)` ; les arguments ont diminué d'une unité. On continue à descendre ainsi jusqu'à tomber sur l'exécution de `pppf_de_f_un (0, 1)` et donc de `f_un (pppf_de_f_un, 0, 1)`. Comme 0 est nul, la réponse finale est *plus petit*. De même, on prouve que plus généralement `plus_petit_point_fixe (f_un, n, m)` renvoie *plus petit* si $n < m$, *égal* si $n = m$ et *plus grand* sinon. Autrement dit, le programme `plus_petit_point_fixe (f_un, _, _)` calcule l'ordre usuel sur $\mathbb{N}$, qui est bien un bon ordre de type ω, plus petit point fixe de f_1.

3.2 Itérons les points fixes

Nous n'avons étudié dans la partie précédente que le plus petit point fixe de f, mais les suivants ne sont-ils pas aussi intéressants ? Concrètement, reprenons une fonction $f : \aleph_1 \to \aleph_1$ croissante et continue, notons λ son plus petit point fixe et intéressons-nous à ce qui se passe « après » λ.

L'idée naturelle, au vu de ce qui précède, consiste à regarder la suite $\left(f^{(n)}(\lambda+1)\right)$. La croissance de f assure la monotonie de cette suite, mais pas forcément la croissance. Le seul cas qui pourrait poser problème est celui où $f(\lambda+1) = \lambda$, ce qui n'est pas exclu par les hypothèses. De fait, nous allons faire une hypothèse supplémentaire sur la fonction f bien plus forte : pour tout ordinal $\alpha \in \aleph_1$, $f(\alpha) \geqslant \alpha$.

Fort de cette nouvelle hypothèse, on déduit que la suite $\left(f^{(n)}(\lambda+1)\right)$ est croissante et converge vers un ordinal $\lambda_1 \in \aleph_1$ qui vérifie $f(\lambda_1) = \lambda_1$. Il s'agit en fait du deuxième point fixe de f. En effet, supposons que $\alpha > \lambda$ soit un point fixe de f. On a alors $\alpha \geqslant \lambda+1$ et donc en appliquant f, il vient $\alpha = f(\alpha) \geqslant f(\lambda+1)$. En itérant le procédé, on obtient pour tout entier n, $\alpha \geqslant f^{(n)}(\lambda+1)$, et donc $\alpha \geqslant \lambda_1$, ce qui est bien ce que l'on voulait.

Le résultat précédent peut se généraliser sans plus d'efforts sous la forme suivante : si $\alpha \in \aleph_1$, le plus petit point fixe de f supérieur ou égal à α est toujours donné par la limite de la suite croissante $\left(f^{(n)}(\alpha)\right)$, bien sûr moyennant les hypothèses faites sur la fonction f.

On pose $\lambda_0 = \lambda$. On a construit λ_1, mais on peut de la même façon construire λ_2, troisième point fixe de f, *etc.* Une fois tous les λ_n construits, pour n entier, on peut définir λ_ω : c'est la borne supérieure des λ_n, la suite des (λ_n) étant de toute façon manifestement croissante. Cette suite est même strictement croissante, ce qui assure à son tour que $\lambda_\omega > \lambda_n$

[10] `pppf` pour Plus Petit Point Fixe.

pour tout entier n. Finalement, la continuité de f implique que λ_ω est un nouveau point fixe ; c'est le plus petit qui soit supérieur à tous les λ_n. Et on peut continuer ainsi, comme le précise la définition suivante :

Définition 1. Soit $f : \aleph_1 \to \aleph_1$ une fonction croissante, continue et vérifiant $f(\alpha) \geqslant \alpha$ pour tout $\alpha \in \aleph_1$. On pose :
- $\mathrm{Fix}\, f(0) = \sup_n f^{(n)}(0)$
- si $\alpha = \beta + 1$, $\mathrm{Fix}\, f(\alpha) = \sup_n f^{(n)}(\mathrm{Fix}\, f(\beta) + 1)$
- sinon, $\mathrm{Fix}\, f(\alpha) = \sup_{\beta<\alpha} \mathrm{Fix}\, f(\beta)$

On construit ainsi à partir de f une nouvelle fonction $\mathrm{Fix}\, f : \aleph_1 \to \aleph_1$ strictement croissante et dont l'image est exactement l'ensemble des points fixes de f. La remarque primordiale, à ce niveau, consiste à dire que la fonction $\mathrm{Fix}\, f$ ainsi définie, est encore continue et vérifie en outre que $\mathrm{Fix}\, f(\alpha) \geqslant \alpha$ pour tout ordinal α. Ces deux points ne posent aucune difficulté, nous laissons le lecteur s'entraîner à les démontrer seul.

On peut donc considérer la fonction des points fixes de $\mathrm{Fix}\, f$, que l'on note $\mathrm{Fix}_2 f$, et puis la fonction des points fixes de cette nouvelle fonction $\mathrm{Fix}_3 f$ et ainsi de suite. Là encore, une fois tous les $\mathrm{Fix}_n f$ définis, il est aisé de donner un sens à $\mathrm{Fix}_\omega f$:

$$\mathrm{Fix}_\omega f(\alpha) = \sup_n \mathrm{Fix}_n f(\alpha)$$

Cette nouvelle fonction est à nouveau croissante, pas forcément strictement. Elle est continue comme il est facile de le voir et elle vérifie encore $\mathrm{Fix}_\omega f(\alpha) \geqslant \alpha$ pour tout ordinal $\alpha \in \aleph_1$. On a tout pour définir un $\mathrm{Fix}_{\omega+1} f$. Et on continue :

Définition 2. Soit $f : \aleph_1 \to \aleph_1$ une fonction croissante, continue et vérifiant $f(\alpha) \geqslant \alpha$ pour tout $\alpha \in \aleph_1$. On pose :
- $\mathrm{Fix}_0 f = f$
- si $\alpha = \beta + 1$, $\mathrm{Fix}_\alpha f = \mathrm{Fix}\,(\mathrm{Fix}_\beta f)$
- sinon, $\mathrm{Fix}_\alpha f = \sup_{\beta<\alpha} \mathrm{Fix}_\beta f$ (*i.e.* on prend la borne supérieure point par point)

Examinons ce que deviennent toutes ces constructions lorsque l'on démarre avec la fonction $f = f_1$, rappelons-le définie par $f(\alpha) = 1 + \alpha$. On a vu que le plus petit point fixe de f est ω. L'ordinal suivant $\omega+1$ est aussi point fixe. Plus généralement, tout ordinal $\alpha \geqslant \omega$ vérifie $f(\alpha) = \alpha$. En effet, un tel ordinal s'écrit $\alpha = \omega + \beta$ et $f(\alpha) = 1 + \omega + \beta = \omega + \beta = \alpha$. On a donc $\mathrm{Fix}\, f(\alpha) = \omega + \alpha$. On trouve ainsi $\mathrm{Fix}\, f = f_2$.

Le plus petit point fixe de f_2 est, on l'a vu également, ω^2. Comme précédemment, on montre que tout ordinal $\alpha \geqslant \omega^2$ est un point fixe de f_2. On en déduit que $\mathrm{Fix}_2 f(\alpha) = \omega^2 + \alpha$. Le plus petit point fixe de cette nouvelle fonction est ω^3 et encore tout ordinal qui lui est supérieur est un nouveau point fixe, ce qui assure $\mathrm{Fix}_3 f(\alpha) = \omega^3 + \alpha$. De la même façon, on aboutit à $\mathrm{Fix}_n f(\alpha) = \omega^n + \alpha$ pour tout ordinal α et tout entier n.

De cela, on peut déduire une expression $\mathrm{Fix}_\omega f$. Pour $\alpha = 0$, on a par définition $\mathrm{Fix}_\omega f(0) = \sup_n \omega^n = \omega^\omega$. Pour $\alpha = 1$, on obtient $\mathrm{Fix}_\omega f(1) = \sup_n \omega^n + 1$ qui vaut encore ω^ω, et ce calcul reste valable pour tout $\alpha < \omega^\omega$ (puisqu'un tel ordinal est forcément inférieur à ω^n pour un certain entier n). Pour connaître la valeur en ω^ω, on peut utiliser la continuité ou, si l'on préfère, refaire le calcul : dans les deux cas, on trouve $\mathrm{Fix}_\omega f(\omega^\omega) = \omega^\omega$. C'est le premier point fixe de $\mathrm{Fix}_\omega f$. Si pour finir $\alpha > \omega^\omega$, alors α est point fixe de $\mathrm{Fix}_n f$ pour tout entier n et donc aussi de $\mathrm{Fix}_\omega f$. On a ainsi décrit la fonction $\mathrm{Fix}_\omega f$ et même la fonction $\mathrm{Fix}_{\omega+1} f$:

$$\mathrm{Fix}_{\omega+1} f(\alpha) = \omega^\omega + \alpha$$

On est maintenant amené à calculer cette dernière fonction : on trouve par des arguments analogues aux précédents $\mathrm{Fix}_{\omega+2} f(\alpha) = \omega^{\omega+1} + \alpha$. Et on peut poursuivre si on y prend plaisir... jusqu'à trouver une formule pour $\mathrm{Fix}_{\alpha}(\beta)$.

Changeons désormais la fonction f de départ et considérons à nouveau $f : \alpha \mapsto \omega^{\alpha}$. L'ordinal $\mathrm{Fix}_1(\alpha)$ est par définition ce que l'on appelle ε_α. De même l'ordinal $\mathrm{Fix}_2(\alpha)$ reçoit le nom de ζ_α et l'ordinal $\mathrm{Fix}_3(\alpha)$ celui de η_α. Remarquez que ces fonctions croissent à une vitesse assez impressionnante. Peut-être, mais on n'est pas obligé d'arrêter la construction à ce niveau. On peut considérer encore la fonction diagonale $\Delta f : \aleph_1 \to \aleph_1$ définie par $\Delta f(\alpha) = \mathrm{Fix}_{\alpha}(\alpha)$. C'est une fonction croissante, continue qui vérifie $\Delta f(\alpha) \geqslant \alpha$ et il est donc légitime de considérer $\mathrm{Fix}\,\Delta f$ mais aussi $\Delta\Delta f$, que l'on note $\Delta^2 f$. On peut itérer la construction et définir $\Delta^n f$ pour tout entier n puis $\Delta^\omega f$ en prenant les bornes supérieures point par point. Finalement on donne un sens à $\Delta^\alpha f$ pour tout ordinal $\alpha \in \aleph_1$.

Là encore, rien ne nous oblige à nous arrêter mais cela commence à devenir un peu répétitif : on pourrait considérer la fonction $\alpha \mapsto \Delta^\alpha f(\alpha)$ et recommencer les constructions avec cette nouvelle fonction. On peut itérer ce procédé un nombre fini ou infini de fois et recommencer ensuite comme il nous plaît un nombre fini ou infini de fois.

3.3 Beaucoup d'ordinaux récursifs

On a expliqué précédemment comment si $f : \aleph_1 \to \aleph_1$ était une fonction (croissante, continue) calculable, on parvenait à écrire un programme calculant son plus petit point fixe. On peut pareillement en écrire un qui calcule son deuxième plus petit point fixe. Le décrire n'est pas quelque chose de vraiment facile : il s'agit d'introduire une nouvelle quantité ∞ (ou -1 ou ce que l'on veut) qu'il faut gérer, tout le problème étant là. En réalité, cela fonctionne exactement comme pour ε_1 sauf qu'il faut expliquer comment on doit modifier les choses pour gérer la nouvelle quantité introduite. Dans le cas de ε_1, il s'agissait de dire à la routine `suite` de tenir compte des ∞ et de faire correspondre le nombre ∞ à la suite de longueur 1, (∞) mais bien sûr ce n'est pas toujours cette transformation qu'il faut faire.

Comme pour ε_α, on peut en outre écrire des méta-méta-programmes (qui n'en sont plus vraiment car ils doivent également modifier le comportement des programmes qu'ils appellent) qui calculent sur $\mathbb{N}$ des bons ordres de type $\mathrm{Fix}\, f(\alpha)$ dès lors que α est récursif. En appliquant ces pseudo-méta-méta-programmes aux pseudo-méta-méta-programmes précédents, on obtient un moyen de calculer $\mathrm{Fix}_2 f(\alpha)$, pour α récursif, et en itérant ainsi on arrive à calculer tous les $\mathrm{Fix}_n(\alpha)$ (pour α toujours récursif, évidemment). Il est même possible avec de nouvelles modifications d'écrire un programme[11] qui calcule $\mathrm{Fix}_\omega f(\alpha)$ pour α récursif et même $\mathrm{Fix}_\alpha f(\beta)$ pour α et β récursifs. Ainsi la fonction diagonale Δf est calculable et rien encore ne nous empêche de continuer.

Ce n'est pas ainsi que l'on arrivera aux bouts de nos peines et il nous faut une approche nouvelle pour essayer de cerner ces ordinaux récursifs, c'est le but du chapitre suivant.

4 Les ordinaux dénotables

4.1 La définition

Un *système de notations* est la donnée d'une fonction $\nu : \mathbb{N} \to \aleph_1$ et de trois programmes désignés par les lettres `t`, `p` et `s`. Aucune hypothèse particulière n'est faite sur la simple

[11] On ne compte plus le nombre de « pseudo » et de « méta » désormais.

fonction ν ; on ne suppose même pas qu'elle est définie partout. On dit que l'entier n est un nom de l'ordinal α si $\nu(n) = \alpha$. Remarquons encore une fois que la fonction ν n'est par exemple pas supposée injective : un même ordinal peut donc être nommé de plusieurs façons différentes.

Les programmes `t`, `p` et `s` ont les prototypes suivants :

```
sorte t (int n);
int p (int n);
suite s (int n);
```

Une variable de type `sorte` ne peut prendre que quatre valeurs qui sont *nul*, *successeur*, *limite* ou *erreur*. Le type `suite`, lui, correspond à un programme de type `int -> int`, c'est-à-dire à un programme `P` dont le prototyte est :

```
int P (int n);
```

Donnons à présent les rôles des programmes `t`, `p` et `s`. Le programme `t` (pour *test*) prend en argument un entier n et doit répondre *nul* si n est un nom de l'ordinal 0, *successeur* si n est le nom d'un ordinal successeur, et *limite* si n est le nom d'un ordinal limite. Dans les autres cas (*i.e.* si n n'est le nom d'aucun ordinal, *i.e.* si la valeur $\nu(n)$ n'est pas définie), il peut soit répondre *erreur*, soit boucler indéfiniment ; il n'y a aucune contrainte à ce niveau.

Le programme `p` (pour *prédécesseur*) prend en argument un entier n. Si `t` renvoie *successeur* sur l'entrée n, `p` doit calculer un entier qui est le nom d'un prédecesseur de l'ordinal nommé n ; cela signifie que l'on doit avoir la formule suivante :

$$\nu(\mathtt{p}(n)) + 1 = \nu(n)$$

Dans le cas où `t` ne renvoie pas *successeur* sur l'entrée n, aucune contrainte n'est imposée sur le comportement de `p`. Il peut répondre un entier arbitraire ou même boucler infiniment.

Le programme `s` (pour *suite*) prend en argument un entier n. Son comportement n'est spécifié que si sur l'entrée n, `t` répond *limite*. Dans ce cas, `s` doit fournir (le code d')un programme noté `s_n` de prototype :

```
int s_n (int i);
```

Ce programme doit calculer une suite α_i strictement croissante d'ordinaux et convergeant vers $\alpha = \nu(n)$. Formellement, on doit avoir les deux conditions suivantes :

$$\begin{aligned} \nu(\mathtt{s\_n}(i+1)) &> \nu(\mathtt{s\_n}(i)) \\ \sup_i \nu(\mathtt{s\_n}(i)) &= \nu(n) \end{aligned}$$

Tout cela forme la définition d'un système de notations.

Un système de notations est dit *injectif* si la fonction ν l'est, c'est-à-dire si tout ordinal de $\aleph_1$ a au plus un nom.

Un ordinal sera dit *dénotable* s'il existe un système de notations (pas forcément injectif) qui lui donne un nom. Arrêtons-nous une minute pour se convaincre que tous les ordinaux auxquels on a pu donner un nom (même un nom baroque comme $\Delta^{\varepsilon_0} f(\eta_0)$ où par exemple $f : \alpha \mapsto \omega^\alpha$) sont dénotables. En fait, précisément la façon dont on a procédé pour nommer tel ordinal fournit *presque* directement un système de notations qui le nomme.

Terminons ce paragraphe par une remarque simple. Si ν (qui vient avec ses trois programmes) est un système de notations tel que décrit précédemment, alors l'ensemble des ordinaux nommés (*i.e.* l'image de ν) est un segment initial de $\aleph_1$. Cela signifie que si ν attribue un nom à l'ordinal α, alors il attribue aussi un nom à tout ordinal $\beta < \alpha$. Pour prouver cela, on constate que si ν attribue un nom à $\alpha+1$, il en attribue nécessairement au moins un à α (puisqu'il doit exister un programme `p` qui calcule ce nom). En contraposant, si α n'est pas nommé par ν, il en est de même de $\alpha + 1$. Pareillement, on prouve que si pour toute suite α_i convergeant vers α ordinal limite, il existe un i tel que α_i ne soit pas nommé par ν, alors α non plus n'est pas nommé. Autrement dit, si α est un ordinal limite et s'il existe $\beta < \alpha$ tel que ν ne nomme aucun ordinal compris strictement entre β et α, alors ν ne nomme pas non plus α.

Ces propriétés dégagées, on considère le plus petit ordinal non nommé par ν, disons α et on prouve par induction sur β que $\alpha + \beta$ n'est jamais nommé par ν. Pour $\beta = 0$, c'est vrai par hypothèse. Si c'est vrai pour β, d'après la première propriété énoncée dans le paragraphe précédent, c'est aussi vrai pour $\beta + 1$. Pour finir si β est un ordinal limite, alors $\alpha + \beta$ l'est aussi et $\alpha < \alpha + \beta$. En outre, par hypothèse d'induction, ν ne nomme aucun ordinal compris strictement entre α et $\alpha + \beta$; il ne nomme donc pas non plus $\alpha + \beta$. Ceci achève notre preuve.

4.2 Un système de notation universel

On définit dans ce paragraphe un système de notation $S : \mathbb{N} \to \aleph_1$ particulier. On se focalise dans un premier temps sur la fonction S, laissant pour après la description des programmes `t`, `s` et `p`.

La définition de S se fait par induction sur les ordinaux : on définit des fonctions partielles S_α sur un certain sous-ensemble de $\mathbb{N}$ en imposant que S_β soit un prolongement de S_α si $\alpha \leqslant \beta$. Dans ces conditions, on est capable de définir S de la façon suivante. Soit un entier n. S'il existe un $\alpha \in \aleph_1$ tel que $S_\alpha(n)$ soit défini, alors d'après ce que l'on a dit précédemment, $S_\beta(n)$ est défini pour β suffisamment grand et sa valeur ne dépend alors pas de β. C'est cette valeur commune que l'on définit comme étant $S(n)$. Si au contraire, $S_\alpha(n)$ n'est défini pour aucun α, on dit de même que $S(n)$ n'est pas défini.

On définit S_0 en posant :

$$\begin{array}{cc} & S_0(1) = 0 \\ \forall n \neq 1 & S_0(n) \text{ non défini} \end{array}$$

Si $\alpha = \beta + 1$, on définit S_α en fonction de S_β. Bien sûr pour tous les entiers n sur lesquels S_β est défini, on pose $S_\alpha(n) = S_\beta(n)$. Et, si n est un entier tel que $S_\beta(n) = \beta$, on définit $S_\alpha(2^n) = \beta + 1 = \alpha$. Pour que cela soit légitime, il faut justifier que S_β n'était pas déjà défini en 2^n. Cela se montre par induction en même temps que cette définition. Nous laissons au lecteur le soin de rédiger proprement cette récurrence transfinie.

Supposons maintenant que α soit un ordinal limite. Si $S_\beta(n)$ est défini pour un certain $\beta < \alpha$, alors sa valeur ne dépend de β pourvu qu'il soit suffisamment grand, et on pose $S_\alpha(n) = S_\beta(n)$. Sinon, on regarde *tous* les programmes P de type `suite` qui calculent une suite strictement croissante de noms d'ordinaux de limite α. À chacun de ces programmes `P`, on associe un certain entier $\mathrm{cod}(\texttt{P})$ défini comme le nombre dont l'écriture en base 2 est le code source du programme. On pose alors $S_\alpha\left(3^{\mathrm{cod}(\texttt{P})}\right) = \alpha$ pour tous les programmes `P` évoqués précédemment. Il reste à voir, pour que cela ait un sens, que les fonctions S_β

($\beta < \alpha$) ne sont définies sur aucun des entiers $3^{\mathrm{cod}(\mathtt{P})}$, ce que nous laissons encore une fois au lecteur.

On comprend alors ce que doivent faire les programmes `t`, `p` et `s`. Sur l'entrée n, `t` répond *nul* si $n = 1$, il répond *successeur* si n est une puissance de 2 autre que 1 et il répond *limite* si n est une puissance de 3 autre que 1. Dans tous les autres cas, `t` répond *erreur*. Le programme `p`, quant à lui, vérifie que son entrée est une puissance de 2 et, le cas échéant, retourne l'exposant. De même, le programme `s` vérifie que son entrée est une puissance de 3 et, le cas échéant, retourne le code du programme qui correspond à l'exposant.

Tout cela nous fournit un système de notations. Soulignons que bien entendu, ce système ne nomme évidemment pas tous les ordinaux de $\aleph_1$ (pour des raisons simples de cardinalité). Autrement dit, il existe un ordinal à partir duquel la construction précédente s'arrête. Bien sûr, il ne peut s'agir d'un ordinal successeur. C'est en fait le plus petit ordinal limite pour lequel il n'existe aucun programme capable de calculer une quelconque suite strictement croissante convergeant vers cet ordinal.

Cet ordinal auquel on s'arrête est étonnamment précisément le plus petit ordinal non dénotable. Autrement dit, le système de notations que l'on vient de définir nomme au moins autant d'ordinaux que n'importe quel autre système de notations. C'est en ce sens que l'on peut dire qu'il est *universel*. Notez par ailleurs que l'existence du plus petit ordinal non dénotable n'est *a priori* pas claire : ce n'est pas parce qu'un système de notations particulier ne peut atteindre tous les ordinaux de $\aleph_1$ que l'on peut en déduire l'existence d'ordinaux qui ne sont atteints par *aucun* système de notations.

Nous allons prouver la propriété (universelle) suivante : pour tout système de notations $\nu : \mathbb{N} \to \aleph_1$ (venant donc avec les programmes `t_nu`, `p_nu` et `s_nu`), il existe une fonction partielle $\varphi : \mathbb{N} \to \mathbb{N}$ calculable[12] faisant commuter le diagramme suivant :

$$\begin{array}{ccc} \mathbb{N} & \xrightarrow{S} & \aleph_1 \\ {\scriptstyle\varphi}\uparrow & & \| \\ \mathbb{N} & \xrightarrow{\nu} & \aleph_1 \end{array}$$

c'est-à-dire tel que $\nu = S \circ \varphi$. En langage courant, cela signifie que l'on peut « passer » du système de notations universel S à ν par une fonction calculable, et ce pour tout système ν.

Remarquons tout de suite que le diagramme précédent prouve que S nomme au moins autant d'ordinaux que ne le fait ν comme on l'a annoncé.

Voyons rapidement comment l'on démontre l'affirmation précédente et pour cela décrivons le programme `phi` calculant la fonction φ. Le pseudo-code suivant est probablement plus clair qu'une longue explication :

[12] On rappelle que cela signifie qu'il existe un programme (donc de type `suite`) qui prend en entrée un entier n et qui doit retourner $\varphi(n)$. Si φ n'est pas défini sur l'entier n, le programme peut avoir le comportement qu'il désire.

```
int phi (int n) {
  t = t_nu (n);
  if (t == ERREUR) return 0;
  if (t == NUL) return 1;
  if (t == SUCCESSEUR) return 2^(phi (p_nu (n)));
  if (t == LIMITE)
    Renvoyer 3 puissance le code du programme suivant[13] :
    int S_n (int i) { return phi (s_nu (n)(i)); };
}
```

Le fait que ce programme calcule bien ce que l'on souhaite est passablement clair. Le point plus délicat est de justifier que ce programme termine, les appels récursifs n'étant pas tout simples. Cependant, on remarque que les appels à `phi` se font à chaque fois sur des entiers qui correspondent à des noms d'ordinaux strictement inférieurs. Le fait qu'il n'existe pas de suites infinies strictement décroissantes d'ordinaux permet alors de conclure.

4.3 Un système de notations universel amélioré

Il est possible de modifier quelque peu la définition précédente et de définir en même temps une relation d'ordre partiel sur l'ensemble des noms attribués correspondant peu ou prou à l'ordre sur les ordinaux nommés.

On procède par induction. On construit simultanément $|\cdot|_0 : \mathbb{N} \to \aleph_1$ (la fonction qui nomme les ordinaux[14]) et un ordre sur $\mathbb{N}$ noté par le symbole $<_0$.

Tout d'abord, on donne le nom 1 à l'ordinal 0. Autrement dit, on pose $|1|_0 = 0$.

Si $\alpha = \beta + 1$ et si x est un nom de β (*i.e.* $|x|_0 = \beta$), on pose comme précédemment, $|2^x|_0 = \alpha$ et on décrète en outre que $z <_0 2^x$ pour tout entier z qui vérifiait déjà $z \leqslant_0 x$.

Finalement, supposons que α soit un ordinal limite. Dans ce cas, on considère les programmes `P` de type `suite` qui calculent une suite strictement croissante d'ordinaux convergeant vers α mais en se restreignant cette fois-ci aux suites strictement croissantes pour l'ordre $<_0$, c'est-à-dire aux programmes `P` tels que :

$$\forall i \in \mathbb{N}, \quad \mathtt{P}\,(i) <_0 \mathtt{P}\,(i+1)$$

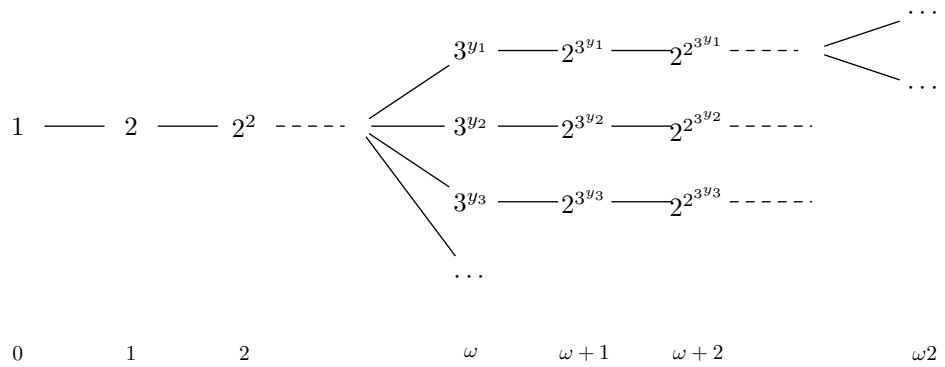
Dans ce cas, comme précédemment, si `P` est un tel programme, on donne le nom $3^{\mathrm{cod}(\mathtt{P})}$ à l'ordinal α, *i.e.* on pose $\left|3^{\mathrm{cod}(\mathtt{P})}\right|_0 = \alpha$. En outre, on rajoute la relation $z <_0 3^{\mathrm{cod}(\mathtt{P})}$ pour tout entier z qui était tel que $z <_0 \mathtt{P}\,(i)$ pour un certain entier i.

Là encore, définir les programmes qui accompagnent la fonction de notation se fait sans problème.

Bien évidemment, le simple fait que $|\cdot|_0$ soit un système de notation prouve qu'il ne nomme pas plus d'ordinaux que S. La question est de savoir s'il en nomme autant. Avant tout, examinons la tête de l'ordre $<_0$. Elle peut être décrite par le schéma suivant :

[13] Oui, il faut faire attention à l'appel récursif de `phi` mais ce n'est pas très compliqué de s'en sortir : on peut par exemple rajouter au code `S_n` celui de `phi`.

[14] Cela signifie que l'on note pour x entier, $|x|_0$ l'ordinal qui porte le nom x.



La dernière ligne correspond à l'ordinal nommé par les entiers écrits au-dessus. Les y_i désignent les codes des programmes qui énumèrent une suite d'entiers strictement croissante au sens de $<_0$, nommant une suite d'ordinaux (strictement croissante donc) convergeant vers ω. Dans notre cas, il s'agit de programmes qui calculent une suite d'entiers dont les termes sont tous une tour de puissances de 2, tour qui devient de plus en plus haute. Évidemment, on imagine qu'il y a énormément de façon de coder une telle chose, et donc que ω a beaucoup de noms.

La propriété fondamentale de α (qui se constate sans surprise sur l'arbre esquissé précédemment) est que si l'on se donne α un ordinal et x un nom de α, alors le sous-ensemble de $\mathbb{N}$ formé des éléments strictement inférieurs à x (au sens de $<_0$) est bien ordonné par $<_0$ et le type de bon ordre est précisément α.

Fermons cette parenthèse et démontrons que le système de notations $|\cdot|_0$ nomme tous les ordinaux dénotables. On aimerait procéder comme précédemment mais il est ici impossible de reprendre directement le programme `phi` du paragraphe précédent : la suite renvoyée par le programme `s_nu` est par hypothèse une suite strictement croissante d'ordinaux, mais rien n'assure que les noms fournis forment une suite strictement croissante d'entiers au sens de $<_0$; on pourrait avoir 3^{y_1} puis $2^{3^{y_2}}$ qui ne sont pas comparables au sens de $<_0$. Pour s'en sortir, il faut être un peu plus malin.

La ruse consiste à définir l'addition sur les noms. Plus précisément, on commence par écrire un programme qui prend en argument deux entiers x et y qui sont respectivement les noms des ordinaux α et β, et qui calcule un entier $z = x +_0 y$, nom d'un ordinal γ, le tout tel que :

1. $\alpha + \beta = \gamma$
2. $x \leqslant_0 z$

Le programme qui implémente ce cahier des charges se fait de façon simple et récursive. Si on désigne par `t_0`, `p_0` et `s_0` les programmes venant avec la fonction de notation $|\cdot|_0$, on peut par exemple écrire :

```
int somme (int x, int y) {
  tx = t_0 (x); ty = t_0 (y);
  if (tx == ERREUR || ty == ERREUR) return 0;
  if (ty == NUL) return x;
  if (ty == SUCCESSEUR) return 2^(somme (x, p_nu (y)));
  if (ty == LIMITE)
    Renvoyer 3 puissance le code du programme suivant :
    int S_n (int i) { return somme (x, s_nu (y)(i)); };
}
```

Dans ces conditions, la première contrainte est bien vérifiée : on a seulement recopier la définition par induction de la somme définie sur les ordinaux. Et, la deuxième l'est également comme on le vérifie sans mal. Cela fait, le programme `phi` peut s'écrire :

```
int phi (int n) {
  t = t_nu (n);
  if (t == ERREUR) return 0;
  if (t == NUL) return 1;
  if (t == SUCCESSEUR) return 2^(phi (p_nu (n)));
  if (t == LIMITE)
    Renvoyer 3 puissance le code du programme suivant :
    int S_n (int i) {
      return phi (somme (S_n (i-1), s_nu (n)(i)));
    };
}
```

Dans ce dernier code, la suite générée par la programme `S_n` est bien strictement croissante au sens de $<_0$. Le problème devient que la suite calculée n'a plus aucune raison de converger vers l'ordinal souhaité. En effet, si le programme `s_nu (n)` renvoie une suite d'entiers correspondant aux ordinaux $\alpha_0, \ldots, \alpha_i, \ldots$, le nouveau `phi` renvoie un programme[15] qui calcule la suite d'ordinaux (β_i) (où par définition $\beta_i = \alpha_0 + \ldots + \alpha_i$) qui n'a aucune raison de converger vers α.

Mais cela n'est pas important. La suite (β_i) converge vers un ordinal β qui a la vertu d'être plus grand que α. Ainsi en reprenant les notations du paragraphe précédent (ce que l'on fait d'ailleurs déjà depuis longtemps), on n'a certes pas l'égalité mais une inégalité persiste, à savoir :

$$\varphi\left(|x|_0\right) \geqslant \nu\left(x\right)$$

Ainsi si α est nommé par ν, il existe un ordinal $\beta \geqslant \alpha$ qui est nommé par $|\cdot|_0$. Cela suffit pour assurer que $|\cdot|_0$ atteint tous les ordinaux dénotables.

4.4 Ordinaux dénotables et ordinaux récursifs

Le théorème est le suivant :

Théorème 1. *Tout ordinal dénotable est récursif.*

Essayons juste d'expliquer globalement comment l'on prouve une telle chose. Dans la partie précédente, on a défini, sur une partie de $\mathbb{N}$, un ordre $<_0$, qui était *presque* un bon ordre dont on pouvait contrôler le type. Si on y réfléchit, c'est donc très proche de ce que l'on cherche à faire. Précisément si α est un ordinal dénotable de nom x, pour calculer un bon ordre de type α, on aimerait écrire le programme suivant :

```
ordre ordinal_x (int n, int m) {
  if (! n <0 x) return ERREUR;
  if (! m <0 x) return ERREUR;
  if (n <0 m) return PLUS_PETIT;
  if (m <0 n) return PLUS_GRAND;
  return EGAL;
}
```

[15] Plutôt 3 puissance le code dudit programme.

Le symbôle « `!` »correspond à la négation logique. Ainsi le premier test est vérifié si (et seulement si) n n'est *pas* plus petit que x pour l'ordre $<_0$. Ainsi, on continue l'exécution du programme si et seulement si $n <_0 x$. On constate directement que le programme précédent calcule l'ordre $<_0$ sur le sous-ensemble de $\mathbb{N}$ formé des entiers $n <_0 x$. Or on sait que cet ordre est un bon ordre de type α. Le programme `ordinal_x` remplit donc *a priori* sa fonction.

Il y a cependant un écueil : il ne semble pas évident d'écrire une routine qui implémente la relation d'ordre $<_0$ (et ainsi les quatre tests que fait le programme `ordinal_x` demandent une justification). De fait, il y a une subtilité : on ne peut pas écrire un programme qui prend en entrée deux entiers n et m et répond systématiquement si n et m sont comparables pour l'ordre $<_0$ et le cas échéant lequel est le plus petit. Cependant, on peut écrire un programme qui prend en entrée deux entiers n et m et donne en retour le plus petit des deux *dans le cas où ils sont comparables*. Dans le cas contraire, le comportement du programme n'est pas spécifié, et typiquement, il ne donnera jamais de réponse. Le point clé consiste à remarquer que cette dernière version est suffisante pour ce que l'on souhaite faire.

Commençons plus modestement par écrire une routine qui, étant donné un entier M, liste tous les entiers inférieurs ou égaux (pour l'ordre $<_0$) à M. Par exemple, si l'on note toujours `t_0`, `p_0` et `s_0` les programmes associés à la fonction de notation $|\cdot|_0$, on peut écrire :

```
void plus_petits (int M) {
  printf ("%d\n", M);

  t = t_0 (M);
  if (t == SUCCESSEUR) plus_petits (p_0 (M));
  if (t == LIMITE) {
    i = 0;
    while (true) { plus_petits (s_0 (M)(i)) &; i++; }
  }
}
```

Explicitons linéairement le code qui précède. Le premier terme, `void` (pour *vide*), signifie simplement que le programme `plus_petits` ne renvoie aucun résultat. La seconde ligne, maintenant, est une syntaxe plutôt contre-intuitive qui se contente d'afficher la valeur de `M` à l'écran. Si vous n'aimez pas, vous pouvez lire `print M` à la place. Il ne reste plus qu'à expliquer le « `&` ». Ce n'est pas une syntaxe correcte ni du `C`, ni du `C++`, mais cela fait longtemps que l'on a oublié l'espoir de faire un programme qui compile. Cette syntaxe est empruntée aux éditeurs de commandes, ce que l'on appelle les `shell`. Elle signifie qu'on lance l'exécution de ce qui précède (en l'occurrence `plus_petits (s_0 (M)(i))`) en parallèle, c'est-à-dire que l'on ne suspend pas pendant ce temps l'exécution du programme courant comme c'est le cas habituellement.

À part cela, il n'y a pas de problème à comprendre le fonctionnement de la procédure : pour afficher les entiers inférieurs ou égaux (au sens de $\leqslant_0$) à M, on teste deux cas. Si M est le nom d'un ordinal successeur, on commence par afficher M et on appelle récursivement la procédure pour afficher les entiers inférieurs ou égaux au prédécesseur de x. Si M est limite, on affiche M et on rappelle récursivement la procédure pour afficher les entiers inférieurs ou égaux à chacun des termes d'une suite strictement croissante dont la borne supérieure est M, le tout toujours au sens de $<_0$.

Il est maintenant possible de modifier quelque peu ce programme pour qu'il n'affiche plus une liste mais pour qu'il fasse un test :

```
bool plus_petit (int n, int m) {
  if (n == m) return true;

  t = t_0 (m);
  if (t == SUCCESSEUR) return plus_petit (p_0 (m));
  if (t == LIMITE) {
    i = 0;
    while (true) { return plus_petit (s_0 (m)(i)) &; i++; }
  }
}
```

Ce dernier programme teste si n est plus petit que m. Le type `bool` est un type booléen[16] : il ne peut prendre que deux valeurs qui sont `true` (pour vrai) et `false` (pour faux). Il faut préciser ce qui se passe lorsque l'on arrive à un `return` : convenons que dans ce cas, le premier `return` rencontré dans une procédure arrête cette procédure en renvoyant la valeur qui suit le `return` et arrête de fait toutes les autres procédures qu'elle avait lancée et qui sont encore en cours d'exécution, et ce de façon récursive.

Remarquons, plus intéressant, que ce programme répond `true` dans le cas où $n \leqslant_0 m$, mais boucle infiniment si ce n'est pas le cas. On peut donner une ultime amélioration :

```
ordre compare (int n, int m) {
  if (n == m) return EGAL;
  if (plus_petit (n, m) &) return PLUS_PETIT;
  if (plus_petit (m, n) &) return PLUS_GRAND;
}
```

Ce dernier programme répond *égal* si $n = m$, *plus petit* si $n <_0 m$ et *plus grand* si $m <_0 n$. Toutefois, si n et m ne sont pas comparables pour la relation d'ordre $<_0$, il boucle infiniment. Et comme nous l'avons déjà dit, cela nous suffit pour la procédure `ordinal_x`. Avec les routines que l'on vient d'introduire, elle peut se réécrire sous la forme :

```
ordre ordinal_x (int n, int m) {
  if (compare (n, x) != PLUS_PETIT) return ERREUR;
  if (compare (m, x) != PLUS_PETTT) return ERREUR;
  return compare (n, m);
}
```

Vérifions qu'elle calcule bien un bon ordre de type α, si x est un nom de α. En effet, si n et m ne sont pas tous les deux plus petits que x, le programme soit boucle (si n (resp. m) et x ne sont pas comparables) soit répond *erreur*, ce qui est bien ce que l'on veut. Maintenant, si à la fois, n et m sont plus petits que x, ils sont comparables et donc la troisième ligne ne peut pas boucler. En conclusion, ce programme calcule bien un bon ordre de type α. Le théorème 1 est démontré !

Il faut noter que la réciproque est aussi vraie : tout ordinal récursif est dénotable. L'idée, que nous n'allons pas détailler, est de construire à partir d'un bon ordre sur $\mathbb{N}$ de type α, un système qui atteint non pas α, mais mieux $\omega \cdot \alpha$. Bien sûr, cela est plus fort, mais permet aussi de se poser beaucoup moins de questions avec les ordinaux successeurs et les ordinaux limites. Tout cela n'est pas extrêmement difficile.

[16] En réalité, un tel type n'existe pas en `C`. Peu importe.

En résumé, les ordinaux récursifs sont exactement les mêmes que les ordinaux dénotables. Les définitions aussi bien des uns que des autres ne permettent pas vraiment de les cerner, mais simplement en observant leur définition il est facile de se convaincre que ces ordinaux peuvent être très grands.

Bibliographie commentée

Ce texte se trouve à la limite entre deux théories présentées généralement de façon séparée, du moins lorsqu'elles sont introduites : la théorie des ordinaux qui s'inscrit souvent dans le cadre de la théorie des ensembles et l'étude des infinis d'une part et la théorie de la récursivité étudiée principalement en informatique d'autre part.

Comme expliqué dans le texte les ordinaux apparaissent comme une vaste généralisation de la notion d'entier. On aimerait continuer à compter après l'infini. Toutefois, la formalisation rigoureuse n'est pas immédiate et bien que notre exposé donne le cadre de cette formalisation, il n'insiste pas vraiment sur les détails techniques. Le lecteur, soucieux d'en savoir plus et de mieux comprendre les constructions faites ici, pourra se reporter dans un premier temps à [1], ou pour plus de détails à [2].

La théorie de la recursivité ou de la calculabilité, quant à elle, étudie principalement les fonctions, disons de $\mathbb{N}$ dans $\mathbb{N}$ qui sont calculables par un ordinateur. Grossièrement, une fonction est dite *calculable* s'il existe un programme qui prend en argument un entier et calcule l'image de cet entier par la fonction en question. On peut montrer par un argument de cardinalité immédiat qu'il existe des fonctions qui ne sont pas calculables, mais par contre il est bien plus difficile d'en exhiber une car souvent la façon dont on définit la fonction donne un algorithme de calcul.

La notion de programme est donc essentielle dans la théorie de la recursivité. Il existe évidemment une définition abstraite d'un programme incarnée par ce que l'on appelle les *machines de Turing*. Mais, si l'on n'a jamais fait de programmation, il ne faut pas commencer par lire ces définitions, mais plutôt par acheter un livre de Basic, de Pascal ou de `C`[17] et manipuler avec un ordinateur pour s'imprégner un peu des concepts fondamentaux. Si taper sur un clavier ne plaît pas, on peut par la suite abandonner, mais nous pensons que c'est une étape nécessaire avant de se lancer dans la théorie abstraite de la récursivité.

Les livres de programmation sont nombreux et relativement bons dans l'ensemble. Citons par exemple pratiquement toute la collection O'Reilly, et en particulier [3], ou peut-être plus agréable pour débuter [4] ou [5]. Notons également que la collection O'Reilly est en grande partie disponible en ligne en anglais ([6]). La théorie de la récursivité est souvent introduite dans des manuels de cours pour ingénieurs, par exemple dans [7] ou encore [8]. Un traité bien plus complet, mais bien plus difficile, mais dans lequel la notion d'ordinal récursif et d'ordinal dénotable est étudiée est [9].

Références

[1] W. Sierpinski, *Leçons sur les nombres transfinis*, Ed. Jacques Gabay

[2] J.L. Krivine, *Theorie des ensembles*, Cassini, Paris, 1998

[3] P. Prinz et U. Kirch-Prinz, *C précis et concis*, O'Reilly

[4] C. Delannoy, *Le livre du C, premier langage*, Eyrolles, 2003

[5] C. Delannoy, *Programmer en langage C*, Eyrolles, 2003

[6] `http://www.oreilly.com/catalog/prdindex.html`

[7] J.M. Autebert, *Calculabilité et décidabilité, une introduction*, Masson, 1992

[17] Évitez par contre de commencer par le `C++` qui introduit des difficultés considérables ou par le caml dont l'esprit est totalement différent et souvent moins accessible dans un premier temps.

[8] J.P. Azra, B. Jaulin, *Récursivité*, Gauthier-Villars, Paris, 1973

[9] H. Rogers Jr, *Theory of Recursive Functions and Effective Computability*, McGraw-Hill, New-York, Toronto, London, *etc.*, 1967