Codage de Gödel

On transforme les symboles, les formules et même les preuves en des entiers. On expliquera ensuite les grandes lignes de la preuve du premier théorème d'incomplétude.

1. Codage des symboles, formules, preuves

1.1. Motivation

Le but est de coder chaque formule par un entier, afin de permettre un traitement mathématique et algorithmique. Une analogie parlante se fait avec le code ASCII qui code un caractère par un entier ('A' : 65, 'B' : 66, ...). Pour coder un texte (une chaine de caractères), il suffit de juxtaposer les entiers ('BAC' : 66/65/67). Comme un ordinateur ne code pas les entiers par les chiffres de 0 à 9 mais seulement avec les symboles 0 et 1, nos caractères ASCII sont écrits sur 8 bits :

A : 01000001
B : 01000010
Z : 01011010

Pour une chaine de caractères il suffit de juxtaposer les codes de chaque caractère en un seul entier.

'BAC' : 010000100100000101000011

Le décodage est facile et commence par séparer le code par paquets de 8 bits.

Il y a une distinction importante à faire entre un nombre et la représentation de ce nombre :

$$7, \quad \text{VII}, \quad \text{sept}, \quad 111_2, \quad \text{SSSSSSS}(0)$$

sont différentes représentations du même entier désigné par '7' (ici en chiffres arabes, romains, en toutes lettres, en écriture binaire et dans le langage $\mathcal{L}_0$ de l'arithmétique de Peano). C'est une distinction bien connue en informatique. Il y a un aspect philosophique profond à donner un nom aux choses, même si le nom ne change pas la chose elle-même. Ainsi, tout comme le mot *table* n'a pas quatre pieds, les symboles arithmétiques servent uniquement de support formel pour représenter les entiers.
Dans la suite, ce qui est vraiment fondamental c'est l'existence d'un codage et d'un décodage. Les rouages internes ne sont pas si importants et varient d'un livre à l'autre. De toute façon, on verra que ce codage est absolument inutilisable en pratique.

1.2. Coder les symboles

On considère le langage $\mathcal{L}_0 = (0, S, +, \times)$ de la logique du premier ordre pour l'arithmétique de Peano. Il faut coder les symboles du langage, les connecteurs, les quantificateurs et les variables $(x_i)_{i\geqslant 1}$.
On choisit (arbitrairement) le codage suivant (voir le livre de Peter Smith) :

$\neg$	$\wedge$	$\vee$	$\rightarrow$	$\leftrightarrow$	$\forall$	$\exists$	$=$	$($	$)$	0	S	$+$	$\times$
1	3	5	7	9	11	13	15	17	19	21	23	25	27

x_1/x	x_2/y	x_3/z	x_4	$\cdots$	x_i
2	4	6	8	$\cdots$	$2i$

Par exemple, le code du symbole « $\exists$ » est $c = 13$. Le symbole du code $c = 21$ est « 0 » (la constante zéro du langage). La variable x_i est codée par l'entier $2i$.
Pour décoder un code, il suffit de lire le tableau de bas en haut :

- si c est impair, chercher à quel symbole il correspond,
- si c est pair, le symbole est la variable $x_{c/2}$.

Remarques :

- Pour certaines preuves on peut limiter le nombre de connecteurs, par exemple à $(\neg, \rightarrow, \forall)$ ou bien à $(\neg, \vee, \forall)$.
- Il est important pour le décodage de ne pas avoir de code qui vaut 0.

1.3. Coder une formule

On considère un mot du langage $\mathcal{L}_0$, c'est donc une suite finie de symboles. Cela peut être une formule « $\forall S(x) = y$ » ou une suite de symboles qui n'a pas de sens « $S(\forall) \rightarrow \neg \times 0$ ».
On va noter $c_1, c_2, c_3, \ldots$ les codes des symboles de ce mot (lu de gauche à droite). Ces codes seront les exposants dans un produit de nombres premiers :

$$2^{c_1} \times 3^{c_2} \times 5^{c_3} \times \cdots$$

On note $\pi_1 = 2$, $\pi_2 = 3$, $\pi_3 = 5, \ldots$ la suite ordonnée des nombres premiers. On peut ainsi formaliser notre construction :

- On part d'un mot (ou d'une formule) $\mathcal{P}$ qui est une suite de symboles $a_1, a_2, \ldots, a_l$ de $\mathcal{L}_0$.
- À chaque symbole a_i, on associe son code c_i.
- Le ***nombre de Gödel*** $n(\mathcal{P})$ de la formule $\mathcal{P}$ est :

$$\boxed{n = \prod_{i=1}^{l} \pi_i^{c_i}}$$

Remarques. C'est généralement un très grand nombre. Cela s'applique aussi à un mot quelconque de $\mathcal{L}_0$, même si ce n'est pas une formule.

> **Exemple.**
> - $\mathcal{P}$: « $\neg(x = 0)$ »
> — $[a_1, \ldots, a_6] = [\neg, (, x, =, 0,)]$
> — $[c_1, \ldots, c_6] = [1, 17, 2, 15, 21, 19]$
> — $n(\mathcal{P}) = 2^1 \cdot 3^{17} \cdot 5^2 \cdot 7^{15} \cdot 11^{21} \cdot 13^{19}$
> - « 2 » n'existe pas dans $\mathcal{L}_0$, son écriture dans $\mathcal{L}_0$ est « $SS0$ » dont le nombre de Gödel est $2^{23} \cdot 3^{23} \cdot 5^{21}$.

1.4. Décoder

Le décodage est basé sur la décomposition en facteurs premiers des entiers (aussi appelé le principe fondamental de l'arithmétique) et en particulier sur l'unicité de cette décomposition.

Théorème 1.
Pour tout entier $n \geqslant 1$, il existe un entier $l \geqslant 1$ et des entiers $\alpha_1, \ldots, \alpha_l \geqslant 0$ tels que :

$$n = \pi_1^{\alpha_1} \times \pi_2^{\alpha_2} \times \cdots \times \pi_l^{\alpha_l}$$

De plus, les α_i sont uniques.

Ainsi, pour décoder un nombre de Gödel n, c'est-à-dire retrouver la formule encodée, on commence par décomposer n en produit de facteurs premiers. Les exposants $c_1, \ldots, c_l$ sont les codes qui redonnent un à un les symboles de la formule.
Il n'y a qu'une seule façon de décoder un nombre. D'un point de vue mathématique, cela signifie que le codage est injectif :

$$\text{Si } \; n(\mathcal{P}) = n(\mathcal{Q}) \; \text{ alors } \; \mathcal{P} = \mathcal{Q}.$$

> **Exemple.**
> $n = 35\,872\,267\,500$
> - Décomposition en facteurs premiers : $n = 2^2 \times 3^{15} \times 5^4$.
> - Les exposants donnent les codes $[2, 15, 4]$.
> - Chaque code se décode en un symbole $[x, =, y]$.
> - La formule de n est donc « $x = y$ ».

1.5. Coder les preuves

On souhaite aussi coder une preuve entière par un seul entier. Une preuve est ici une suite finie de formules $(\mathcal{P}_1, \mathcal{P}_2, \ldots, \mathcal{P}_L)$. Comment coder une preuve ? Chaque $\mathcal{P}_j$ est codée par son nombre de Gödel $n_j = n(\mathcal{P}_j)$ Ensuite, on associe à la preuve son ***super-nombre de Gödel*** :

$$N = 2^{n_1} \cdot 3^{n_2} \cdots \pi_L^{n_L}$$

c'est-à-dire :

$$\boxed{N = \prod_{j=1}^{L} \pi_j^{n(\mathcal{P}_j)}}$$

Remarques :
- C'est un nombre très très grand. On ne le calculera jamais en pratique.
- On ne mélangera jamais les nombres de Gödel et les super-nombres de Gödel. Ce sont deux codages distincts. On saura au préalable si un entier donné désigne un nombre de Gödel ou un super-nombre de Gödel.

Comment décoder un super-nombre de Gödel ?
- On décompose N en produit de facteurs premiers.
- On en extrait la liste des exposants $(n_1, \ldots, n_L)$.
- On décode chaque nombre de Gödel n_j, en une formule $\mathcal{P}_j$.
- On obtient la liste de formules $(\mathcal{P}_1, \ldots, \mathcal{P}_L)$.

2. Algorithme de décodage

Nous avons réussi à coder les symboles, les formules, les preuves par des entiers. Il va falloir montrer que les opérations associées sur les entiers sont « suffisamment simples », ce qui correspondra à la notion de *fonction récursive primitive* discutée dans les chapitres suivants. Pour l'instant on va aborder cette problématique d'un point de vue algorithmique, afin de mieux appréhender le travail futur.
On commence par les opérations de codage/décodage. Que ces opérations soient « suffisamment simples » signifie qu'on peut les programmer en utilisant uniquement des boucles « pour » (*for*) bornées (le nombre d'itérations est borné en fonction de l'entrée). Cela signifie qu'on s'interdit les boucles « tant que » (*while*) pour lesquelles on ne sait pas à l'avance quand sera atteinte la condition d'arrêt.

2.1. Décoder le code d'un symbole

Il est facile d'écrire un algorithme `decode_code_symbole(c)` qui en entrée reçoit un entier c (le code valide d'un symbole) et qui en sortie renvoie le symbole $a \in \mathcal{L}_0$ correspondant (voir la section précédente). Il n'y a même pas besoin d'une boucle « pour ».

Algorithme (`decode_code_symbole(c)`).
- — Entrée : c un entier.
  — Sortie : a un symbole de $\mathcal{L}_0$.
- Si $c = 1$, $a \leftarrow$ « $\neg$ ».
- Si $c = 3$, $a \leftarrow$ « $\wedge$ ».
- ...
- Si $c = 27$, $a \leftarrow$ « $\times$ ».
- Si c est pair, $a \leftarrow x_{c/2}$.
- Renvoyer a.

2.2. Décoder un nombre de Gödel

Lemme 1.
Soit $n = \pi_1^{\alpha_1} \times \cdots \times \pi_l^{\alpha_l}$ *avec* $\alpha_i \geqslant 1$ $(i = 1, \ldots, l)$.
- $l \leqslant \log_2(n) \leqslant n$
- $\alpha_i \leqslant \log_2(n) \leqslant n \quad (i = 1, \ldots, l)$

Démonstration. On rappelle que par définition $\log_2(2^n) = n$. Comme $2^n \geqslant n$ alors $n \geqslant \log_2(n)$.
- $n = \pi_1^{\alpha_1} \cdots \pi_l^{\alpha_l} \geqslant \pi_1 \times \cdots \times \pi_l \geqslant 2^l$ donc $\log_2(n) \geqslant l$.
- $n = \pi_1^{\alpha_1} \cdots \pi_l^{\alpha_l} \geqslant \pi_i^{\alpha_i} \geqslant 2^{\alpha_i}$ donc $\log_2(n) \geqslant \alpha_i$.

□

La boucle de l'algorithme suivant est bornée par $K = \lfloor \log_2(n) \rfloor$. Le lecteur pointilleux qui considère que calculer $\log_2(n)$ est une opération compliquée remplacera $\log_2(n)$ par n.

Algorithme (`decode_nombre_godel(n)`).
- — Entrée : n un nombre de Gödel.
  — Sortie : un mot/formule $[a_1, \ldots, a_l]$.
- `liste` $\leftarrow$ `[ ]` (liste vide).
- Pour k allant de 1 à $K = \lfloor \log_2(n) \rfloor$:
  $c \leftarrow$ `exposant`(n, π_k)

Si $c > 0$, ajouter `decode_code_symbole(c)` à `liste`.
- Renvoyer `liste`.

On laisse au lecteur le soin d'écrire l'algorithme de décodage d'un super-nombre de Gödel N.

2.3. Nombres premiers

On s'autorise les opérations arithmétiques de base $+$, $-$, $\times$, $/$. Cependant, il faut s'assurer que les opérations sur les nombres premiers sont faciles à réaliser.
Tout d'abord, le test de divisibilité `est_divisible(n,d)` qui décide si d divise n serait facile à programmer avec une boucle « tant que ». Voici l'algorithme avec une boucle « pour ».

Algorithme (`est_divisible(n,d)`).
- — Entrée : deux entiers n, d.
  — Sortie : Vrai ssi d divise n.
- Pour i allant de 1 à n :
  $n \leftarrow n - d$
  Si $n = 0$, renvoyer Vrai (et terminer la boucle).
  Si $n < 0$, renvoyer Faux (et terminer la boucle).

On en déduit un algorithme `est_premier(n)` qui renvoie Vrai lorsque n est un nombre premier et Faux sinon, en testant s'il existe un diviseur d parmi $2, 3, \ldots, n-1$.
Ensuite, on sait récupérer l'exposant de p dans la décomposition d'un entier n en facteurs premiers. Par exemple `exposant`$(2^a 3^b 5^c, 3)$ est b.

Algorithme (`exposant(n,p)`).
- — Entrée : n un entier, p un nombre premier.
  — Sortie : l'exposant e de p dans n.
- $e \leftarrow 0$.
- Pour k allant de 1 à $K = \lfloor \log_2(n) \rfloor$:
  Si `est_divisible(n, p)` :
  $e \leftarrow e + 1$
  $n \leftarrow n/p$
  Sinon quitter la boucle.
- Renvoyer e.

Enfin, on a partout utilisé la liste $\pi_1 = 2$, $\pi_2 = 3, \ldots$ de tous les nombres premiers. Il faut vérifier qu'on sait calculer cette liste aussi loin que nécessaire. On programme une fonction `prochain_premier(n)` qui renvoie le premier nombre premier plus grand que n. Le postulat de Bertrand (prouvé par Tchebychev) affirme qu'il existe toujours un nombre premier entre n et $2n$. Il suffit donc de tester la primalité de $n+1, \ldots, 2n$. On obtient alors facilement une fonction `liste_premiers(K)` qui renvoie la liste de tous les nombres premiers inférieurs à K.

3. Opérations sur les formules

Nous allons maintenant justifier qu'à partir d'un nombre de Gödel n, on peut « facilement » dire si la formule correspondante vérifie une propriété. Par exemple, la formule est-elle bien définie ? Est-elle une formule fermée ? Est-elle réduite à une variable ? Et pour un super-nombre de Gödel N : encode-t-il une preuve correcte ?

3.1. Formules

- `est_variable(n)` teste si la formule de nombre de Gödel n est une des variables x_i.
- `est_terme(n)` teste si la formule de nombre de Gödel n est un terme de $\mathcal{L}_0$. On rappelle que les termes sont la constante 0, les variables x_i et si t_1 et t_2 sont des termes alors $S(t_1)$, $t_1 + t_2$, $t_1 \times t_2$ sont aussi des termes.
- `est_formule(n)` renvoie Vrai lorsque n encode une formule valide du langage $\mathcal{L}_0$ (« $x = \forall$ » n'est pas valide par exemple).
- `est_formule_fermee(n)` renvoie Vrai si n encode une formule sans variable libre.

Voici par exemple l'algorithme pour tester si n encode un terme de $\mathcal{L}_0$.
Algorithme `est_terme(n)` :

- — Entrée : un entier n.
  — Sortie : Vrai ssi n encode un terme de $\mathcal{L}_0$.
- $\mathcal{P} \leftarrow$ `decode_nombre_godel(n)`
- Si $\mathcal{P} =$ « 0 », renvoyer Vrai
- Si $\mathcal{P}$ est une variable, renvoyer Vrai
- Si $\mathcal{P}$ s'écrit « $S(t)$ », renvoyer `est_terme(t)`
- Si $\mathcal{P}$ s'écrit « $t_1 + t_2$ », renvoyer `est_terme(t_1)` ET `est_terme(t_2)`
- Si $\mathcal{P}$ s'écrit « $t_1 \times t_2$ », renvoyer `est_terme(t_1)` ET `est_terme(t_2)`
- Sinon, renvoyer Faux

C'est une fonction récursive, dont la profondeur est inférieure à la longueur de $\mathcal{P}$, donc inférieure à $\log_2(n)$. On pourrait en écrire un algorithme avec des boucles en utilisant des piles.

3.2. Preuve

Il est fondamental qu'on sache décider efficacement si une preuve donnée est valide ou pas. C'est le rôle de la fonction `est_preuve(N,n)`.

- Soit N super-nombre de Gödel représentant une suite de formules $\mathcal{P}_1, \ldots, \mathcal{P}_L$.
- Soit n un nombre de Gödel représentant une formule $\mathcal{Q}$.
- La fonction `est_preuve(N,n)` renvoie Vrai ssi $\mathcal{P}_L = \mathcal{Q}$ et que $\mathcal{P}_1, \ldots, \mathcal{P}_L$ forment une preuve (une suite de formules obtenues par les règles d'inférence).

$$\begin{array}{ll} & \mathcal{P}_1 \\ & \vdots \\ & \mathcal{P}_{L-1} \\ \vdash & \mathcal{P}_L = \mathcal{Q} \end{array}$$

Ici il est plus facile de supposer que les preuves sont écrites dans le système de Hilbert, dans lequel on a des axiomes et des règles d'inférence de la forme $\mathcal{P}_a \to \mathcal{P}_b$. Il n'y a pas d'hypothèses, ni sous-hypothèses, comme dans nos preuves en déduction naturelle. Pour obtenir $\mathcal{P} \vdash \mathcal{Q}$ on prouve en fait $\mathcal{P} \to \mathcal{Q}$.
Ainsi, on vérifie facilement :

- soit $\mathcal{P}_i$ est un axiome,
- soit $\mathcal{P}_i$ se déduit de $\mathcal{P}_a$ et $\mathcal{P}_b$ avec $a, b < i$ où $\mathcal{P}_a$ est de la forme $\mathcal{P}_b \to \mathcal{P}_i$.
- Si $\mathcal{P}_1, \dots, \mathcal{P}_L$ forment une preuve valide et que $\mathcal{P}_L = \mathcal{Q}$ alors on renvoie Vrai.

3.3. Valeur entière vs numéral

On revient sur la différence entre un nombre et l'écriture de ce nombre, avec deux nouvelles notations :
- $\ulcorner \mathcal{P} \urcorner$: le nombre de Gödel de $\mathcal{P}$,
- le ***numéral*** $\overline{n}$: c'est l'écriture d'un entier n dans le langage $\mathcal{L}_0$.

$\mathcal{P}$	$\longleftrightarrow$	$n = \ulcorner \mathcal{P} \urcorner$	$\longleftrightarrow$	$\overline{n} = \overline{\ulcorner \mathcal{P} \urcorner}$
formule de $\mathcal{L}_0$		nombre de Gödel, dans $\mathbb{N}$		numéral de n, dans $\mathcal{L}_0$

Exemple :

$$\mathcal{P} : x = 0 \quad \longleftrightarrow \quad n = \ulcorner \mathcal{P} \urcorner = 2^{\cdots} \cdot 3^{\cdots} \cdot 5^{\cdots} \quad \longleftrightarrow \quad \overline{n} = \overline{\ulcorner \mathcal{P} \urcorner} = \underbrace{SS \dots S}_{n \text{ occurrences}} 0$$

On écrira souvent $\mathcal{Q}(n)$, au lieu de $\mathcal{Q}(\overline{n})$ car il ne peut pas y avoir de confusion. Ainsi, lorsque l'on écrit « `est_formule(n)` », si l'on veut vraiment un algorithme dans le langage $\mathcal{L}_0$, n doit être écrit sous la forme du numéral $\overline{n} = SS \dots S0$.

3.4. Concaténation

Deux formules (ou mots) $\mathcal{P}$ et $\mathcal{Q}$ juxtaposées donnent la ***concaténation*** $\mathcal{P} * \mathcal{Q}$. Par exemple « $S(x)$ » $*$ « $+$ » $*$ « y » donne « $S(x) + y$ ». Cette opération semble immédiate, cependant elle est plus délicate à écrire lorsque nos formules sont représentées par des entiers :
- $\ulcorner S(x) \urcorner = 2^a \cdot 3^b \cdot 5^c \cdot 7^d$
- $\ulcorner + \urcorner = 2^e$
- $\ulcorner y \urcorner = 2^f$
- $\ulcorner S(x) + y \urcorner = 2^a \cdot 3^b \cdot 5^c \cdot 7^d \cdot 11^e \cdot 13^f$

Formellement, si $n = \prod_{i=1}^{l} \pi_i^{\alpha_i}$ et $m = \prod_{j=1}^{l'} \pi_j^{\beta_j}$ alors :

$$n * m = n \times \prod_{j=1}^{l'} \pi_{j+l}^{\beta_j}$$

3.5. Substitution

Considérons une formule $\mathcal{P}$ de variable libre x, que l'on note généralement $\mathcal{P}(x)$. Quand on écrit $\mathcal{P}(n)$ c'est en fait la substitution $\mathcal{P}[n/x]$. Il faut maintenant écrire cela dans le langage $\mathcal{L}_0$, c'est-à-dire qu'il nous faut une fonction `substitution`$(\ulcorner \mathcal{P} \urcorner, \ulcorner x \urcorner, n)$ qui renvoie $\overline{\ulcorner \mathcal{P}(\overline{n}) \urcorner}$, le nombre de Gödel de la substitution.

> **Exemple.**
> - $\mathcal{P}(x)$: « $y = x + 0$ », de nombre de Gödel $\ulcorner \mathcal{P}(x) \urcorner = 2^{\cdots} \cdot 3^{\cdots} \cdot 5^{\cdots} \cdot 7^{\cdots} \cdot 11^{\cdots}$.
> - $\mathcal{P}(2)$: « $y = 2 + 0$ », mais « 2 » n'est pas un symbole du langage, il faut écrire $\overline{2} = SS0$.
> - Ainsi, $\mathcal{P}(\overline{2})$: « $y = SS0 + 0$ », c'est bien une formule de $\mathcal{L}_0$.
> - $\ulcorner \mathcal{P}(\overline{2}) \urcorner = 2^{\cdots} \cdot 3^{\cdots} \cdot 5^{\cdots} \cdot 7^{\cdots} \cdot 11^{\cdots} \cdot 13^{\cdots} \cdot 17^{\cdots}$ est le nombre de Gödel de la substitution.
> - Il ne reste plus qu'à transformer $\ulcorner \mathcal{P}(\overline{2}) \urcorner$ en son numéral $\overline{\ulcorner \mathcal{P}(\overline{2}) \urcorner}$.

C'est un bon exercice, pas trop difficile, d'écrire l'algorithme de substitution en utilisant la concaténation.

4. Preuve du théorème d'incomplétude par diagonalisation

4.1. Le paradoxe de Russell

Nous allons donner l'idée générale de la preuve du premier théorème d'incomplétude. La preuve repose sur le principe de diagonalisation qui prend ici la forme d'auto-référencement, c'est-à-dire une formule qui parle d'elle-même. On commence par une petite analogie avec le paradoxe de Russell, popularisé sous la forme de l'énigme suivante :

Dans un village, le barbier rase tous les hommes qui ne se rasent pas eux-mêmes, et seulement ceux-là. Qui rase le barbier ?

Si on suppose que le barbier ne se rase pas lui-même, alors le barbier se rase lui-même. Et s'il se rase lui-même, il ne doit pas se raser lui-même. C'est inextricable !

La version suivante a servi à Russell pour justifier qu'il ne peut pas exister un ensemble contenant tous les ensembles. Si un tel ensemble $\mathcal{E}$ existait, on pourrait considérer :

$$y = \{x \in \mathcal{E} \mid x \notin x\}$$

Question : Est-ce que $y \in y$?
Remarque : y est bien un ensemble de $\mathcal{E}$, car $\mathcal{E}$ contient tous les ensembles.

- Si $y \in y$ alors par définition de y, $y \notin y$.
- Si $y \notin y$ alors par définition de y, $y \in y$.

On obtient une contradiction, aussi l'ensemble de tous les ensembles ne peut pas exister.

4.2. La diagonalisation et la formule de Gödel

Formule de départ.
Voici la formule que l'on va étudier :

> $\mathcal{P}(x)$: « La formule obtenue à partir de la formule de nombre de Gödel x, en y substituant sa variable libre par x, est non démontrable. »

Bien sûr, dans les chapitres suivants on écrira cette formule de façon purement mathématique.
Pour l'instant, remarquons :

- Pour un entier n, on sait qu'il peut être vu comme le nombre de Gödel d'une formule $\mathcal{Q}$ que l'on saurait retrouver (voir les sections précédentes).
- Si cette formule $\mathcal{Q}$ possède une variable libre, par exemple $\mathcal{Q}(y)$, alors on peut effectuer la substitution $\mathcal{Q}[n/y]$, notée $\mathcal{Q}(n)$.
- Enfin, pour un n donné on peut discuter si la formule fermée $\mathcal{Q}(n)$ est démontrable ou pas.

Il faut maintenant justifier que cette formule $\mathcal{P}(x)$ est bien une formule de notre langage $\mathcal{L}_0$. On a vu que « `est_formule()` » et « `est_preuve()` » sont des algorithmes faciles, ce qui signifie qu'ils pourraient s'exprimer dans le langage $\mathcal{L}_0$ de l'arithmétique de Peano. Mais qu'en est-il de l'affirmation « cette formule est démontrable » ?
On peut définir :

$$\texttt{est\_demontrable(n)} \quad \text{ssi} \quad \exists N\ \texttt{est\_preuve(N, n)}.$$

Le problème principal, d'un point de vue algorithmique, est qu'on n'a pas de borne à priori sur le N, donc un algorithme correspondant utiliserait une boucle « tant que » afin de tester $N = 1$, $N = 2$, ... Si la formule est démontrable, l'algorithme finit par s'arrêter, mais ce n'est pas le cas si elle est non démontrable. C'est donc un mauvais algorithme de notre point de vue.

En revanche ; « $\exists N$ `est_preuve(N, n)` » est bien une formule de notre langage du premier ordre. Ainsi, « cette formule n'est pas démontrable » est bien une formule de $\mathcal{L}_0$. Ce qui fait de $\mathcal{P}(x)$, avec x comme variable libre, une formule de $\mathcal{L}_0$.

Diagonalisation.
Comme $\mathcal{P}(x)$ est une formule de $\mathcal{L}_0$, elle admet un nombre de Gödel. Notons :

$$\boxed{n = \ulcorner \mathcal{P}(x) \urcorner}$$

le nombre de Gödel de la formule $\mathcal{P}$.
La formule de Gödel est alors définie ainsi :

$$\boxed{\mathcal{G} = \mathcal{P}(n)}$$

C'est donc la formule $\mathcal{P}$ évaluée en son propre nombre de Gödel, autrement dit :

$$\boxed{\mathcal{G} = \mathcal{P}(\ulcorner \mathcal{P} \urcorner)}$$

Le théorème d'incomplétude.
Avec cette formulation encore informelle, voici le théorème d'incomplétude et sa preuve.

Théorème 2.
Il existe des formules vraies et non démontrables.

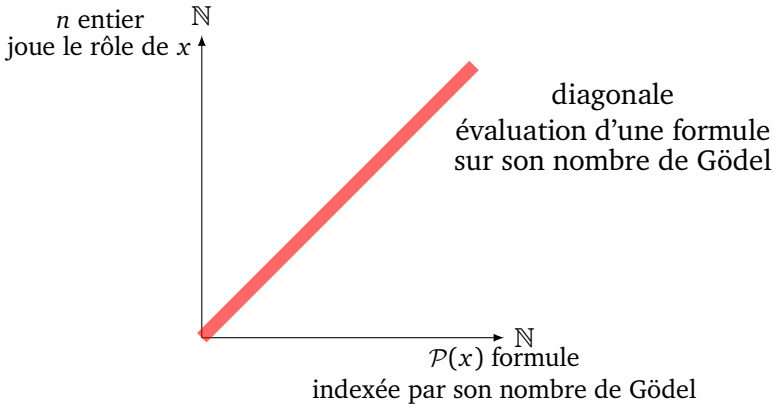
Nous allons montrer que $\mathcal{G}$ est une telle formule. Tout d'abord, $\mathcal{G}$ dit « la formule obtenue à partir de la formule de nombre de Gödel n, en y substituant sa variable libre par n, est non démontrable ». Mais la partie « la formule obtenue à partir de la formule de nombre de Gödel n, en y substituant sa variable libre par n » désigne exactement la formule $\mathcal{P}$ évaluée en n, c'est-à-dire $\mathcal{G} = \mathcal{P}(n)$, car n est le nombre de Gödel de $\mathcal{P}$. Ainsi, la phrase $\mathcal{G}$ dit exactement « $\mathcal{G}$ n'est pas démontrable ».

Démonstration.
- Par l'absurde, supposons $\mathcal{G}$ fausse. Alors $\neg\mathcal{G}$ est vraie, ce qui signifie « $\mathcal{G}$ est démontrable ». Mais par le théorème de validité : si $\mathcal{G}$ est démontrable alors $\mathcal{G}$ est vraie. Contradiction.
- Ainsi, $\mathcal{G}$ est vraie, mais alors cela signifie « $\mathcal{G}$ n'est pas démontrable ».
- Conclusion : $\mathcal{G}$ est vraie, mais n'est pas démontrable.

□

4.3. La diagonale de Cantor

Pourquoi le procédé expliqué ci-dessus s'appelle un procédé de diagonalisation ? C'est facile à comprendre sur le schéma ci-dessous. On a des formules $\mathcal{P}$ (ou $\mathcal{P}(x)$) qui, on l'a vu, sont indexées par des entiers de $\mathbb{N}$ (leur nombre de Gödel). Et on a des entiers $n \in \mathbb{N}$ qui jouent le rôle de x dans l'évaluation. La phrase de Gödel $\mathcal{G}$ est l'application de la formule $\mathcal{P}$ exactement en $n = \ulcorner \mathcal{P} \urcorner$.



D'où vient cette idée de la diagonalisation ? Elle est déjà présente dans le paradoxe de Russell où la question « Est-ce que $x \in y$? » donne sur la diagonale « Est-ce que $y \in y$? ». Cette méthode a surtout été utilisée par Cantor pour prouver que l'ensemble des réels n'est pas dénombrable. Étudions sa preuve.

Théorème 3.
$\mathbb{R}$ n'est pas dénombrable.

Démonstration. Supposons par l'absurde que l'ensemble des réels $[0,1[$ soit un ensemble dénombrable, c'est-à-dire en bijection avec $\mathbb{N}$. En termes plus familiers, cela signifie que tous les réels de $[0,1[$ peuvent être numérotés $x_0, x_1, x_2, \ldots$ c'est-à-dire $[0,1[= \{x_k\}_{k\in\mathbb{N}}$.
On écrit chaque réel de $[0,1[$ par son écriture décimale, par exemple

$$x_0 = 0.189\,335\ldots$$
$$x_1 = 0.772\,004\ldots$$
$$x_2 = 0.191\,542\ldots$$
$$x_3 = 0.053\,477\ldots$$

Cette écriture décimale est unique si on s'interdit l'écriture avec une infinité de 9 à la fin. Par exemple pour $x = 0.123\,999\,999\ldots$ on choisit l'écriture $x = 0.124\,000\,000\ldots$ On remplit un tableau avec les décimales de tous les $(x_k)_{k\geqslant 0}$ (sans le 0 avant la virgule).

⋮						
⋮						
x_3	0	5	3	**4**	7	7
x_2	1	9	**1**	5	4	2
x_1	7	**7**	2	0	0	4
x_0	**1**	8	9	3	3	5
	Décimales					

⋮					
⋮					
	0	5	3	**1**	7	7
	1	9	**2**	5	4	2
	7	**1**	2	0	0	4
	2	8	9	3	3	5

Ensuite, on change le chiffre numéro k de x_k selon la règle suivante :

- si ce chiffre est 1, on le change en 2,
- et si ce chiffre est différent de 1, on le change en 1.

Avec ce procédé on peut lire les décimales d'un nouveau réel $x' \in [0,1[$ sur la diagonale. Sur notre dessin ce serait $x' = 0.2121\ldots$
Mais par construction x' ne peut pas être l'un des x_k car x' et x_k ont leur décimale numéro k distincte. C'est une contradiction car $\{x_k\}_{k\in\mathbb{N}}$ devait énumérer tous les réels de $[0,1[$. Conclusion : $[0,1[$ n'est pas dénombrable et par conséquent $\mathbb{R}$ non plus. □