

Résumé de LINFO1123

compilation du 8 mars 2023

Thomas Debelle

Juin 2023

Table des matières

1	Concepts	3
1.1	Ensemble	3
1.2	Ensemble énumérable	4
1.3	Cantor	4
2	Programmes calculables	6
2.1	Les algorithmes	6
2.1.1	Calculabilité	6
2.2	Fonction calculable	6
2.2.1	Ensemble récursif	6
2.3	Thèse de Church-Turing	7
2.4	Non calculabilité	8
2.4.1	Problème de l'arrêt	8
2.5	Insuffisance des fonctions totales	9
2.5.1	Théorème de Hoare-Allison	9
2.5.2	Interpréteur	10
2.6	Extension des fonctions partielles	10
3	Questions Test d'entrée	11
3.1	TP1	11
3.2	TP2	11
3.3	TP3	12

Préface

Bonjour à toi !

Cette synthèse recueille toutes les informations importantes données au cours, pendant les séances de tp et est amélioré grâce au note du Syllabus. Elle ne remplace pas le cours donc écoutez bien les conseils et potentielles astuces que les professeurs peuvent vous donner. Notre synthèse est plus une aide qui on l'espère vous sera à toutes et tous utiles.

Elle a été réalisée par toutes les personnes que tu vois mentionné. Si jamais cette synthèse a une faute, manque de précision, typo ou n'est pas à jour par rapport à la matière actuelle ou bien que tu veux simplement contribuer en y apportant ta connaissance ? Rien de plus simple ! Améliore la en te rendant [ici](#) où tu trouveras toutes les infos pour mettre ce document à jour. (*en plus tu auras ton nom en gros ici et sur la page du github*)

Nous espérons que cette synthèse te sera utile d'une quelconque manière ! Bonne lecture et bonne étude.

Chapitre 1

Concepts

Dans ce chapitre, on s'intéresse aux ensembles, cardinalité et équipotences de ces derniers

1.1 Ensemble

Un ensemble est un *collection* d'objets, *sans répétition*, ces derniers sont appelés *éléments* de l'ensemble. Donc un ensemble peut être des chiffres, des lettres, il peut être vide symbolisé par *void*. On peut réaliser des opérations dessus, on peut déterminer des *sous-ensembles d'ensemble* donc des ensembles issus d'ensemble. On a également une notion s'appelant le *complément* d'un ensemble dénoté \tilde{A}

Langage

Un *langage* n'est autre qu'un mot ou bien un ensemble de caractères d'une taille fixée. Une chaîne vide est écrite via le caractère " ϵ ". On forme un langage via un *alphabet* qui n'est autre qu'un ensemble de symboles, on le dénote " Σ ". Tout langage est donc une suite de symboles issue de l'*alphabet*. Σ^* correspond à l'ensemble des langages formés via l'alphabet.

Relations

Lorsque nous avons deux ensembles appelés A et B , on peut établir une relation appelée R qui nous donne un sous-ensemble $A \times B$. On peut représenter la relation par une table.

Fonctions

Lorsque nous avons deux ensembles appelés A et B , on peut avoir ce qu'on appelle une *fonction* f . C'est une relation tel que :

$$\exists a \in A : \exists b \in B : \langle a, b \rangle \in f \quad (1.1)$$

Il n'existe pas plus d'un b pour un a . Si pour un a il n'existe pas de b , on dit que $f(a)$ est indéfini et donc $f(a) = \perp$ ou *bottom*.

Propriétés des fonctions

- un *domaine de fonction* ou $\text{dom}(f) = \{a \in A \mid f(a) \neq \perp\}$
- une *image de fonction* ou $\text{image}(f) = \{b \in B \mid \exists a \in A : b = f(a)\}$
- f est dit *fonction totale* si $\text{dom}(f) = A$
- f est dit *fonction partielle* si $\text{dom}(f) \subsetneq A$

- f est **surjectif** ssi $\text{image}(f) = B$ autrement dit, tout élément est associé à minimum 1 élément dans B .
- f est **injectif** ssi $\forall a, a' \in A : a \neq a' \Rightarrow f(a) \neq f(a')$ autrement dit on ne fait correspondre qu'au plus un élément de A dans B .
- f est **bijectif** s'il combine *surjectif* et *injectif*

Intéressons nous aux **extensions** qui est le fait de rajouter une fonction qui ne définit un élément de B pas encore défini.

$$\forall x \in A : g(x) \neq \perp \Rightarrow f(x) = g(x) \quad (1.2)$$

f à la même valeur que g partout où g est défini.

Définition d'une fonction

Comme dit précédemment, une fonction est défini par sa table. On va souvent utiliser une description de la table qui permet que celle-ci soit clair et bien défini. De plus, on a pas besoin de savoir comment calculer ceci.

On peut également définir une table via une fonction ou un algorithme.

1.2 Ensemble énumérable

On dit que 2 ensembles ont le même cardinal (A et B) ssi il existe une bijection entre ces 2 ensembles. Donc chaque élément de A correspond à un élément de B .

On dit d'un ensemble qu'il est dénombrable ssi il est **fini** ou il existe une **bijection** entre l'ensemble \mathbb{N} et cet ensemble.

Exemples

- L'ensemble \mathbb{Z}
- L'ensemble des nombres pairs
- Des paires d'entiers
- L'ensemble des programmes Java

Propriétés

Tout sous-ensemble d'ensemble énumérable est *énumérable*. L'union et l'intersection d'ensembles énumérables est *énumérable*.

En s'intéressant à l'ensemble des programmes informatiques, on se rend compte que c'est une *ensemble énumérable infini*. De plus, les programmes informatiques ne considèrent que des choses *énumérables*.

1.3 Cantor

Le théorème de *Cantor* nous dit que l'ensemble des nombres entre 0 et 1 compris est *non énumérable*.

$$E = \{x \in \mathbb{R} | 0 < x \leq 1\} \quad (1.3)$$

Preuve

Pour prouver cela, on va réaliser une table et on va réaliser une *diagonalisation de Cantor*.

	chiffre 1	chiffre 2	...	chiffre $k + 1$...
x_0	x_{00}	x_{01}	...	x_{0k}	...
x_1	x_{10}	x_{11}	...	x_{1k}	...
...
x_k	x_{k0}	x_{k1}	...	x_{kk}	...
...

Ensuite, on va définir notre nombre de la diagonale qui vaut $d = 0.x_{00}x_{11}...x_{kk}$. De cet valeur, on va créer une valeur d' qui a comme propriété $x_{kk} \neq x'_{kk} \forall k$.

Mais, on doit stocker notre valeur d' dans la table. On la stock à p ce qui donne $d' = 0.x'_{p0}x'_{p1}...x'_{pp}$ mais à cause de la construction de $d = 0.x_{00}x_{11}...x_{pp}$. Par construction, $x'_{pp} \neq x_{pp}$ mais cela ne peut être respecté. Donc, **il n'y a pas** de *bijection* des \mathbb{N} vers cet ensemble. Donc cet ensemble est *non énumérable*.

Autre ensemble non énumérable

- L'ensemble des \mathbb{R} .
- L'ensemble des sous-ensemble de \mathbb{N} .
- L'ensemble des chaines infinies de caractères d'un alphabet fini.
- L'ensemble des *fonctions* de \mathbb{N} dans \mathbb{N} .

Chose intéressante à noter, comme on a une infinité non énumérable de fonctions \mathbb{N} dans \mathbb{N} et un nombre de programme informatique *infini énumérable*. On ne peut résoudre tous les problèmes informatiques donc.

Chapitre 2

Programmes calculables

2.1 Les algorithmes

Un algorithme est un ensemble *d'instructions* qui a pour but de produire un résultat. Donc un algorithme n'est **pas une fonction**. Il *calcule* une fonction. Un algorithme n'est pas forcément un *programme*, il peut être un *organigramme*. C'est un *ensemble fini d'instructions*. C'est une sorte de *calculateur*. Ici, on va considérer nos algorithmes comme *n'ayant pas de limite* de :

- Taille de données
- Taille d'instructions
- Taille de la mémoire, mais on a une utilisation finie.

2.1.1 Calculabilité

Avant de continuer, il faut définir la *calculabilité* des algorithmes car sans *formalisme*, les algorithmes sont non rigoureux, non exploitables.

Ici, on base cette notion sur celle des *programmes informatiques*. (plus intuitif). Ainsi, on possède **2 univers** celui des *programmes informatiques* et celui des *problèmes*. Pour être plus précis, on se base sur le langage **Java** et on se limite au fonction $\mathbb{N} \rightarrow \mathbb{N}$. Ainsi pour les fonctions, on aura **1 entrée** et **1 sortie**. (on peut également généraliser ceci en disant que $\mathbb{N}^n \rightarrow \mathbb{N}$)

2.2 Fonction calculable

Une fonction est dite *calculable* s'il existe un *programme Java* recevant **1 donnée** étant un nombre $\in \mathbb{N}$ et la fonction va nous retourner la *valeur* de $f(x)$ *si* elle est défini.

Si le programme *ne se termine pas* donc pas défini ou erreur d'exécution on dit que $f(x) = \perp$. On définit bien la notion de calculabilité sur *l'existence d'un programme*. on a 2 types de fonctions

1. Fonction *partielle* calculable : on a *parfois* un résultat
2. Fonction *totale* calculable : on peut *toujours* calculé quelque chose.

2.2.1 Ensemble récursif

Maintenant, on va essayer de déterminer la calculabilité sur *un ensemble de fonctions*. Le principe de décision de *calculabilité* est le principe dit *récursif*.

A est **récursif** si il existe un programme *Java* qui recevant n'importe quelle donnée sous forme d'un \mathbb{N} fourni comme résultat :

- 1 si $x \in A$
- 0 si $x \notin A$

Donc on est face à un *algorithme* qui calcule si x est dans A ou non. C'est un algorithme complet et se termine toujours. (attention de ne pas confondre *récuratif* et *récurativité*)

On dit qu'un ensemble d'algorithme est **récurativement énumérable** s'il est *récuratif* sauf qu'il retourne $\neq 1$ $x \notin A$ ou ne se termine pas et qu'on puisse énumérer cet ensemble.

Fonctions caractéristiques

Une fonction caractéristique de $A \subseteq N$ et :

$$X_A : N \rightarrow N : X_A(x) = 1 \text{ si } x \in A \quad (2.1)$$

$$= 0 \text{ si } x \notin A \quad (2.2)$$

C'est une autre manière de déterminer si un ensemble est récuratif si X_A est une fonction *calculable*. On dit qu'une fonction est récurativement énumérable ssi il existe une fonction f calculable ayant pour domaine A . Ou bien, on dit que A est vide *ou* l'image de f est A ayant une fonction f *totale* calculable.

Un **ensemble récurativement énumérable** est un ensemble dont la bijection des N est énumérable et calculable.

Propriétés :

- A récuratif $\Rightarrow A$ récurativement énumérable
- A récuratif $\Rightarrow (N \setminus A)$ récurativement énumérable
- A récuratif $\Rightarrow (N \setminus A)$ récuratif
- A récurativement énumérable et $(N \setminus A)$ récurativement énumérable $\Rightarrow A$ récuratif
- A fini $\Rightarrow A$ récuratif
- $(N \setminus A)$ fini $\Rightarrow A$ récuratif
- A récuratif $\Rightarrow \bar{A}$ récuratif

2.3 Thèse de Church-Turing

Comment démontrer qu'une fonction **n'est pas** calculable.

Les 4 grands points de la thèse :

1. Aucun modèle de la notion de fonction calculable n'est plus puissant que les Machines de Turing (ici Java)
2. Toute fonction calculable (au sens intuitif) est calculable par une machine de Turing (ici Java)
3. Toutes les définitions formelles de la calculabilité connues à ce jour sont équivalentes (Théorème)
4. Toutes les formalisations de la calculabilité établies par la suite seront équivalentes aux définitions connues

On établit que Java à accès à une infinité de mémoire (donc physiquement possible). Ainsi, on a P qui est l'ensemble des programmes Java syntaxiquement corrects, qui reçoivent 1 données *entières* et qui retournent un résultat *entier*.

- P est un ensemble récuratif (infini dénombrable)
- $P = P_0, P_1, \dots, P_k, \dots$ sans répétition donc chaque programme est unique.
- Pour simplifier, $f(k) = P_k$
- f est calculable.
- k et P_k représente le même objet

donc on dit que P_k donne le programme k dans l'ensemble P . on dit que φ_k est la fonction mathématique calculé par P_k . Donc on peut avoir $\varphi_m == \varphi_n$ car réalise le même travail mais sont issues de programmes *différents*. $\varphi_k : N \rightarrow N$.

2.4 Non calculabilité

Pour rappel :

- Nombre de fonctions de $\mathbb{N} \rightarrow \mathbb{N}$ est **non** dénombrable.
- Nombre de programmes Java est dénombrable.

En programmation, on s'intéresse aux fonctions définies de manière finie, donc on a une **infinité dénombrable**. Mais si une fonction est définie de manière finie, peut-elle être calculable ?

2.4.1 Problème de l'arrêt

Une fonction prends 2 paramètres : $\text{halt} : \mathbb{P} \times \mathbb{N}$. P est le numéro du programme et N est son entrée.

$$\text{halt}(n, x) = 1 \text{ si } \varphi_n(x) \neq \perp \quad (2.3)$$

$$= 0 \text{ sinon} \quad (2.4)$$

$$\text{halt}(n, x) = 1 \text{ si l'exécution du } P_n \text{ se termine} \quad (2.5)$$

$$= 0 \text{ sinon} \quad (2.6)$$

On a donc une table finie, mais décrite de manière finie donc bien définie. Peut-on la calculer ?

Preuve par l'absurde

	0	1	...	k	...
P_0	halt(0,0)	halt(0,1)	...	halt(0,k)	...
P_1	halt(1,0)	halt(1,1)	...	halt(1,k)	...
...
P_k	halt(k,0)	halt(k,1)	...	halt(k,k)	...
...

On va sélectionner les valeurs sur la diagonale et stocker cela comme une variable s'appelant "*diag*". On va donc modifier cette valeur et est représenté par "*diag_{mod}*" qui inverse chaque nombre. (donc $0 \rightarrow 1$ et $1 \rightarrow 0$) Donc *diag_{mod}* est calculable sous l'hypothèse que la fonction "halt" l'est.

Donc, il existe un programme Java qui calcule cette "*diag_{mod}*" qu'on trouvera en ligne d . Mais à cause de cela, $\text{diag}_{\text{mod}}(d) \neq \text{diag}(d)$ donc ne peut exister par *définition*.

En conclusion, la fonction "halt" **n'est pas** calculable.

Conclusion

- Aucun algorithme ne permet de déterminer pour tout programme P_n et donnée x si $P_n(x)$ se termine ou non
- Seule possibilité serait d'avoir un langage de programmation dans lequel tous les programmes se terminent. La fonction halt est alors calculable pour les programmes de ce formalisme
- halt non calculable ne signifie pas que pour un programme k donné, $\text{halt}(k, x)$ est non calculable

Pour le premier point, on ne peut séparer le soucis en 2 algorithmes car on ne peut changer de programmes selon l'input. Un algorithme donne le **bon résultat** en fonction du résultat.

On dit que halt n'est pas calculable dans le sens où il n'existe pas d'algorithmes **généraux**.

Exemple non-récursif

$$Halt = \{(n, x) \mid halt(n, x) = 1\} \quad (2.7)$$

$$= \{(n, x) \mid P_n(x) \text{ se termine}\} \quad (2.8)$$

$$K = \{n \mid (n, n) \in HALT\} \quad (2.9)$$

$$= \{n \mid halt(n, n) = 1\} \quad (2.10)$$

$$= \{n \mid diag(n) = 1\} \quad (2.11)$$

$$= \{n \mid P_n(n) \text{ se termine}\} \quad (2.12)$$

$$(2.13)$$

Donc K et $HALT$ **ne sont pas** récursifs mais sont récursivement énumérable car si un élément n'appartient pas à K ou $HALT$, il va boucler mais fournir la bonne solution s'il appartient à ces ensembles.

De plus K est la diagonale de $HALT$.

\overline{HALT} n'est pas récursivement énumérable car on a pas de moyen de prouver qu'un probable n'appartient pas à cet ensemble. pareil pour \overline{K}

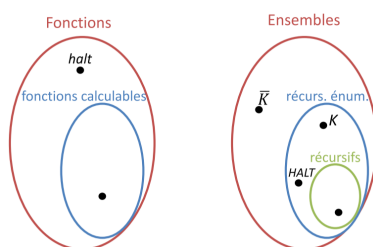


FIGURE 2.1 – Schématisation des fonctions et ensembles de fonctions

Le fait de ne pas pouvoir calculer "halt" nous pose soucis et va ouvrir tout un pan de soucis et limitations.

De plus, on peut faire face à des ensembles **co-récursivement** énumérable. En effet, il existe des ensembles co-récursivement énumérable tel que : \overline{K} et tous les ensembles *récursifs*.

2.5 Insuffisance des fonctions totales

Pourquoi est-ce utile d'avoir un programme qui tourne en *boucle* ? N'avons-nous pas besoin de fonctions qui donnent un résultat précis tout le temps donc *totale* ?

Imaginons que nous créons un *langage de programmation* qui a que des fonctions totales. Donc, **Halt** est calculable et on a une réponse pour toutes fonctions. Halt serait la fonction constante 1.

Notre langage Q est calculable donc ayant un interpréteur calculable.

2.5.1 Théorème de Hoare-Allison

Donc en résumé de notre langage Q :

- L'interpréteur de ce programme est calculable
- La fonction *halt* est totale et correspond à la fonction constante de 1.
- Mais l'interpréteur **n'est pas** calculable *dans* Q .

	0	1	...	k	...
Q_0	interpret(0,0)	interpret(0,1)	...	interpret(0,k)	...
Q_1	interpret(1,0)	interpret(1,1)	...	interpret(1,k)	...
...
Q_k	interpret(k,0)	interpret(k,1)	...	interpret(k,k)	...
...

La colonne Q correspond à l'ensemble des programmes et la ligne de nombre correspond aux entrées de chaque programme.

Tous les programmes se terminent donc *jamais* \perp . On sélectionne la diagonale :

$$diag(n) = interpret(n, n) \quad (2.14)$$

$$diag_{mod}(n) = interpret(n, n) + 1 \quad (2.15)$$

$$Q_l = diag_{mod} \quad (2.16)$$

Et donc, on voit facilement que à la ligne l il y aura un souci avec $diag$ et notre entrée à Q_l qui n'est autre que $diag_{mod}$. en effet $diag_{mod}(l) \neq diag(l)$. Donc la fonction $interpret$ n'est **pas** calculable en Q .

Le *théorème* nous dit donc que : Si un langage de programmation (non trivial) ne permet que le calcul de fonctions totales, alors :

- l'interpréteur de ce langage n'est pas programmable dans ce langage
- il existe des fonctions totales non programmables dans ce langage
- ce langage est **restrictif**

Donc si on peut faire un interpréteur d'un langage dans son langage, la fonction $halt$ n'est pas totale. Donc c'est soit programmable par lui-même soit fonction totale de $halt$.

Si on veut qu'un langage de programmation permette la programmation de toutes les fonctions totales calculables, alors ce langage doit également permettre la programmation de fonctions non totales.

De plus, si on avait une fonction qui regarde si des fonctions sont totales, cela pose problème. Cela est impossible et donc cette fonction $tot(n)$ n'est pas récursif.

2.5.2 Interpréteur

Pour qu'un formalisme soit assez puissant, il faut que ce dernier arrive à programmer son propre interpréteur.

$$\exists z \forall n, x : \varphi_z(n, x) = \varphi_n(x) \quad (2.17)$$

Avec φ_z qu'on appelle la fonction universelle. et P_z est le programme universel. Par convention, on appelle $\theta(n, x)$ la **fonction universelle**.

2.6 Extension des fonctions partielles

Pour l'instant, nous n'avons vu que des fonctions qui soit donnent le bon résultat soit donne \perp et donc boucle. On va réaliser des *extensions*, c'est-à-dire que nous allons retourner la valeur correcte dans les cas possible et un message ou autre chose pour le reste des entrées.

Un *théorème* nous dit que, *Il existe une fonction partielle calculable g telle qu'aucune fonction totale calculable n'est une extension de g .*

Pour prouver cela, on utilise la fonction $nbstep(n, x)$ qui correspond au nombre d'instruction avec l'arrêt de $P_n(x)$. ($P_n(x) = \perp$) La preuve se fait pas diagonalisation comme avant ([vidéo](#)).

Chapitre 3

Questions Test d'entrée

3.1 TP1

1. Effectivement, il existe une bijection entre les \mathbb{N} et les nombres impairs positifs \rightarrow en somme il existe une fonction qui transforme les \mathbb{N} en impair positif
2. J'imagine qu'il y a une bijection mais je ne vois pas quel formule passant de \mathbb{N} aux impairs existent car c'est le propre des nombres impairs
3. Même raisonnement que la question 1
4. La fameuse formule qui lie \mathbb{N} et \mathbb{Q} car \mathbb{Q} est juste une paire de \mathbb{N}
5. Effectivement, sachant la diagonalisation de Cantor il est simple de le prouver
6. Pour \mathbb{N} dans \mathbb{N} il en existe une infinité et l'ensemble d'arrivé ne change pas grand-chose car on s'intéresse au nombre de fonction.
7. Effectivement, on a un nombre fini de langage donc de mot. Cela est dû grâce à l'alphabet fini et la longueur fixe. Donc on sait énumérer
8. Question typique vu au cours. En effet comme on a une infinité et il n'existe aucune bijection depuis les naturels etc.
9. Même cardinalité = bijection, il ne peut y avoir de bijection entre un ensemble non énumérable et énumérable
10. Une infinité de nombre mais effectivement même cardinalité car tous peuvent être ramené aux naturels.

3.2 TP2

1. Effectivement, on ne doit pas être en capacité de coder l'algorithme pour que la fonction soit calculable.
2. Nombre premier est récursif car on a le crible d'ératosthène.
3. Si un ensemble \mathbf{X} est récursif (donc donne 1 ou 0) alors $\bar{\mathbf{X}}$ l'est également car il inverse les 1 et 0.
4. Si \mathbf{X} est récursivement énumérable (donne 1 ou quelque chose d'autre mais pas 1) et que son opposé $\bar{\mathbf{X}}$ est récursivement énumérable. Alors bien évidemment \mathbf{X} est récursif.
5. Oui, trivial.
6. Non, juste au simple fait que si \mathbf{X} est récursivement énumérable alors $\bar{\mathbf{X}}$ ne peut être récursivement énumérable.
7. Vrai car on a une fonction calculable car ce sont des combinaisons linéaires de calculables.
8. Voir sous-section 2.2.1.

9. Oui car être énumérable c'est dire qu'on peut compter tous les résultats même si ça prend un temps infini.
10. Voir sous-section 2.2.1.

3.3 TP3

1. En effet, si on a que des fonctions totales, on sait qu'on aura toujours une réponses pour n'importe quelle input.
2. Oui, le théorème de *Hoare-Allison* ne dit pas l'inverse
3. Mais si on peut avoir la fonction halt en L, on ne peut avoir sa fonction *interpret*
4. Mais, on peut calculer cette fonction *interpret* avec un langage de programmation qui n'est pas **restrictif**.
5. Effectivement on ne peut pas calculer toutes les fonctions totales avec L.
6. Il existe un langage qui peut calculer sa fonction halt et son interpréteur (le langage vide trivial)
7. Effectivement, ne pas être récursif n'empêche pas d'être récursivement énumérable.
8. Faux exemple : $\overline{\mathbf{K}}$. (voir 2.4.1)
9. Non, on peut imaginer 2 fonctions non récursives qui se "*complètent*" et comblent les lacunes de chacune.
10. Non, on peut imaginer une intersection qui ne comportent que des entrées avec une réponse.