# Calculabilité avancée
# Théorèmes de récursion et d'incomplétude

Kévin Perrot – Aix Marseille Université – hiver 2023-24

## Table des matières



# 1 Théorèmes de récursion

Sources : [9] (chapitre 6.1) et [6].

Pour cette section nous aurons besoin de nous souvenir des points suivants.

Il existe une *énumération des fonctions calculables*. Comme il est d'usage dans la littérature, nous noterons

$$\phi_e \text{ la } e^{\text{ième}} \text{ fonction calculable}$$

(nous avons vu qu'il existe une énumération des MT, ce qui revient au même que d'énumérer les fonctions calculables, puisque les fonctions calculables sont calculées par ces mêmes MT). Le nombre $e$ peut être vu comme la représentation du code d'un programme. On utilisera l'égalité $\phi_e = \phi_{e'}$ lorsque les machines calculent la même fonction.

Il existe un *interpréteur* (une machine de Turing universelle)

$$\phi_u(n, \vec{x}) = \phi_n(\vec{x}),$$

qui prend en entrée la description d'une fonction calculable (le code d'une MT) et reproduit cette fonction (simule cette MT) sur le reste de l'entrée ($\vec{x}$).

**Théorème s-m-n** : il existe une fonction calculable $s : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ telle que

$$\phi_{s(m,n)}(\vec{x}) = \phi_m(n, \vec{x}).$$

Ce théorème dit que l'on peut bêtement [1] coder en dur les arguments. Notons que le programme $s(n, n)$ n'a pas accès à son propre code.

1. la fonction $s$ est calculable, on suit son algorithme pour mettre en dur $n$ dans le code de $m$.

## 1.1 Théorème du point fixe de Kleene

**Théorème 1.** *Pour toute fonction (totale) calculable $h$, il existe un programme $n$ tel que*

$$\phi_n(\vec{x}) = \phi_{h(n)}(\vec{x}).$$

Le théorème du point fixe de Kleene nous dit que pour toute transformation algorithmique $h$ sur les programmes, il existe un programme qui fait la même chose que son transformé. Et comme dit au début de la phrase, c'est vrai pour toute transformation $h$ !

*Démonstration.* Soit le programme $m$ suivant : sur l'entrée $t, \vec{x}$, calculer $s(t,t)$, puis calculer $h(s(t,t))$, puis simuler la machine $h(s(t,t))$ sur l'entrée $\vec{x}$. C'est-à-dire,

$$\phi_m(t, \vec{x}) = \phi_u(h(s(t,t)), \vec{x}).$$

Alors $n = s(m,m)$ est notre point fixe, car (définition de $s$, ci-dessus, définition de $\phi_u$)

$$\phi_{s(m,m)}(\vec{x}) = \phi_m(m, \vec{x}) = \phi_u(h(s(m,m)), \vec{x}) = \phi_{h(s(m,m))}(\vec{x}).$$

□

Intuitivement, on souhaite construire un programme $m$ qui simule $h(m)$. Cependant $m$ ne se connaît pas lui même, et nous sommes tentés d'utiliser $s(m,m)$. Ainsi on souhaite construire un programme $s(m,m)$ qui simule $h(s(m,m))$, c'est-à-dire avec $m$ le programme qui, sur une entrée $t$ (qui se trouvera être $m$ dans $s(m,m)$), simule $h(s(t,t))$.

## 1.2 Quine (*self-replicating programs*)

Le théorème du point fixe de Kleene engendre un corollaire divertissant.

**Définition 2.** *Un* **quine** *est un programme qui affiche à l'écran son propre code source.*

**Théorème 3.** *Tout langage de programmation acceptable*[2] *admet des quines.*

*Démonstration.* Pour un programme $t$, considérons le programme $h(t)$ qui affiche à l'écran le code de $t$. La fonction $h$ est calculable[3]. Appliquons le théorème 1 à la fonction $h$ : il existe un programme $n$ tel que les programmes $h(n)$ et $n$ sont identiques, donc $n$ affiche à l'écran le code de $n$. □

Les *virus informatique* sont des programmes basés sur l'auto-réplication. Dans le but de réaliser cette tâche, un virus peut contenir la construction d'un quine pour reproduire son propre code. Comment construire un tel programme en pratique ? Relisez les preuves, elles sont **constructives**, c'est-à-dire qu'elles prouvent l'existence de programmes possédant certaines propriétés tout en expliquant comment les construire. Par contre il y a besoin d'une machine universelle...

Voici comment construire un quine, composé de deux machines de Turing.
— La machine $A$ écrit le code de la machine $B$ sur le ruban.
— La machine $B$ :

1. lit son entrée $w$ sur le ruban,

2. un langage de programmation est *acceptable* s'il vérifie les points précisés en début de section.
3. on peut écrire un programme $h$ qui prend en entrée le code de $t$, et produit le code d'une machine qui affiche le code de $t$.

2. calcule le code de la machine $W$ qui écrit le mot $w$ sur le ruban,
3. écrit sur le ruban le code la machine $W$ composée avec la machine $w$ (*càd* la machine $W$ dont l'état final est remplacé par l'état initial de $w$).

La définition de $A$ dépend de celle de $B$, mais celle de $B$ ne dépend pas de celle de $A$. On peut donc construire la machine $B$ et obtenir son code, ce qui nous permet de constuire $A$ qui écrit ce code. La composition des machines $A$ et $B$, appelons la $C$, est un quine. En effet, quand on lance la machine $C$ :

— la machine $A$ écrit le code de $B$ sur le ruban, atteint son état final,
— qui est l'état initial de la machine $B$ ($C$ étant la compsition des machines $A$ et $B$), qui s'exécute donc sur l'entrée que $A$ vient de laisser sur le ruban :
1. lit le code de la machine $B$,
2. calcule le code de la machine qui écrit le code de $B$ sur le ruban, qui se trouve être la machine $A$,
3. écrit sur le ruban la composition des machines $A$ et $B$, qui se trouve être $C$.

Lui-même. La construction de $C$ correspond exactement aux démonstrations qui précèdent : la partie $A$ code en dur le code de $B$ en premier argument pour $B$ (donc permet de passer de $B$ à $s(B, B)$).

En Bash on obtient (il ne faut pas utiliser le symbole `'` dans le code de $m$ pour le codage en dur dans la variable `t`, donc pour cela on le place dans la variable `z`) :

```
#!/bin/sh
# s(m,m)
z=\'
t='echo "#!/bin/sh
# s(m,m)
z=\\${z}
t=${z}${t}${z}
# m=h(s(t,t))
${t}"'
# m=h(s(t,t))
echo "#!/bin/sh
# s(m,m)
z=\\${z}
t=${z}${t}${z}
# m=h(s(t,t))
${t}"
```

Un jeu d'initié[4] consiste à trouver le plus petit quine dans son langage préféré... ci-dessous quelques exemples de quine en C, Bash, Haskell, Java, OCaml, Python et en français (des sauts de lignes ont été ajoutés).

```
#include<stdio.h>
main(){char*a="#include<stdio.h>%cmain(){char*a=%c%s%c;printf(a,10,34,a,34);
}";printf(a,10,34,a,34);}
```

4. essayez d'écrire un quine, vous verrez que ce n'est pas facile !

```
z=\' a='z=\\$z a=$z$a$z\; eval echo \$a'; eval echo $a

s="main=putStr (['s', '='] ++ show s ++ [';'] ++ s)";
main=putStr (['s', '='] ++ show s ++ [';'] ++ s)

class Quine{public static void main(String[] args){char n=10;char b='"';
String a="class Quine{public static void main(String[] args)
{char n=10;char b='%c';String a=%c%s%c;System.out.format(a,b,b,a,b,n);}
}%c";System.out.format(a,b,b,a,b,n);}}

(fun s -> Printf.printf "%s %S;;" s s)
 "(fun s -> Printf.printf \"%s %S;;\" s s)";;

a='a=%r;print(a%%a)';print(a%a)

Recopier puis recopier entre guillemets la phrase
« Recopier puis recopier entre guillemets la phrase »
```

## 1.3 Avoir accès à son propre code

On peut appliquer le théorème 1 pour montrer qu'un programme peut avoir accès à son propre code, dans le sens où il suffit de supposer qu'il prend en premier argument son propre code, et alors le théorème 1 nous construit un programme équivalent qui se passe du premier argument (ouf).

**Théorème 4.** *Pour tout $m$, il existe $n$ tel que $\phi_n(\vec{x}) = \phi_m(n, \vec{x})$.*

Utilisation : on construit $m$ dont on suppose qu'elle prend en premier argument son propre code, le théorème 4 nous donne $n$ qui fait la même chose que $m$, avec vraiment son propre code à elle intégré à la place du premier argument (« *filled in automagically* »).

*Démonstration.* Soit un $m$ arbitraire, on considère la fonction calculable $h(t) = s(m, t)$. Le théorème 1 nous donne $n$ tel que $\phi_n(\vec{x}) = \phi_{h(n)}(\vec{x}) = \phi_{s(m,n)}(\vec{x}) = \phi_m(n, \vec{x})$. □

Ainsi on peut considérer qu'une machine peut faire appel à l'instruction

« obtenir mon propre code »

puisqu'en applicant le théorème 4 on obtient un programme équivalent qui a effectivement accès à son propre code. Cette instruction est **très puissante pour construire des preuves d'indécidabilité** : informellement, pour tout programme $f$ qui *soi-disant* décide une propriété d'un programme qui lui est donné en entrée, on peut construire un programme $g$ « pathologique » qui :

1. obtient son propre code $g$,
2. calcule le résultat de $f$ pour l'entrée $g$,
3. fait le contraire.

Alors $f$ se trompe sur l'entrée $g$, donc un tel programme $f$ n'existe pas (démonstration par l'absurde en supposant l'existence de $f$).

**Théorème 5.** *L'arrêt des programmes n'est pas décidable.*

*Démonstration.* Par l'absurde, si on suppose que le programme $h$ décide, étant donné un programme $f$ et une entrée $\vec{x}$, si le calcule de $f$ sur $\vec{x}$ s'arrête, alors on peut construire le programme $g$ qui, sur une entrée $\vec{x}$ :

1. obtient son propre code $g$,
2. calcule $\phi_h(g, \vec{x})$ (on a supposé que $h$ est calculable),
3. si $h$ prédit l'arrêt, alors $g$ entre dans une boucle infinie, sinon $g$ s'arrête.

Alors pour tout $\vec{x}$ le résultat de $\phi_h(g, \vec{x})$ est incorrect, une contradiction. □

**Définition 6.** *Pour une machine $M$, la* taille *de la description de $M$ est le nombre de lettres du mot $\langle M \rangle$ (ou bien, pour une énumération $(\phi_e)_{e\in\mathbb{N}}$ des MT, la taille d'une description de $\phi_e$ est simplement le nombre $e$). Une machine $M$ est* minimale *s'il n'existe pas de machine de plus petite taille équivalente*[5] *à $M$. Soit*

$$MIN = \{\langle M \rangle \mid M \text{ est une MT minimale}\}.$$

*Attention, cette définition dépend de notre encodage $\langle M \rangle$ ou de notre énumération $(\phi_e)_{e\in\mathbb{N}}$.*

**Théorème 7.** *$MIN$ n'est pas récursivement énumérable.*

*Démonstration.* Par l'absurde, si on suppose que le programme $m$ énumère $MIN$, alors on peut construire le programme $f$ qui, sur l'entrée $x$ :

1. obtient son propre code $f$,
2. lance la machine $m$ qui énumère $MIN$, et attend de la voir énumérer le code d'une machine $g$ plus grande que son propre code $f$,
3. simule $g$ sur l'entrée $x$.

Puisque l'ensemble $MIN$ est infini, il énumère nécessairement le code de machines de tailles arbitraires, et le second point termine. On a $f$ qui est équivalente à $g$ mais a un code plus petit, ce qui est en contradiction avec le fait que $m$ ait énuméré $g$. □

# 2 Théorèmes d'incomplétude de Gödel

Avec l'idée de chercher une preuve algorithmiquement et nos développements sur la calculabilité, on peut démontrer le premier théorème d'incomplétude de Gödel. Nos preuves vont employer la notion de calcul, alors que Gödel a fait cela à l'intérieur des systèmes formels.

Sources : surtout [1], et aussi [2, 3, 4, 5, 7, 8].

**Mise en garde :** les développements qui suivent sont plein de subtilités, et on leur fait rapidement « dire » plus de qui est démontré. Ce sont des mathématiques qui parlent des mathématiques : des « métamathématiques » ( ?).

## 2.1 Un peu d'histoire (début du XXe siècle)

Source : bande-dessinnée *Logicomix* [2], très chaleureusement recommandée !

En 1900 les mathématiciens se posent la question [2, page 105 case -1]

5. c'est-à-dire qui reconnaît le même langage ou qui calcule la même fonction.

« qu'est-ce que les mathématiques ? »

— The map of mathematics (https://njbiblio.com/tag/computer-science/).
— Idée : fondements = axiomatisation. Russell-Whitehead [2, page 106 case 1].
— Hilbert est convaincu que « tout ce qui peut être énoncé rigoureusement, peut être répondu logiquement » [2, page 142].
— Exemple de difficulté : le paradoxe de Russell [2, page 152 et page 159 case 2].
— Les *Principia Matematica* : redéfinir toutes les mathématiques à partir de la théorie des ensembles (https://fr.wikipedia.org/wiki/Principia_Mathematica#Preuve_de_1+1=2).

Une nouvelle question émerge alors [2, page 258 case 1]

« Peut-on démontrer tous les théorèmes vrais ? »

— Hilbert est toujours convaincu que « oui » [2, page 267 case -1].
— Et Gödel démontre que « non » [2, de page 269 cases -4 à page 270 case 2].

## 2.2 Définitions et énoncés

**Définition 8.** *Un* **système formel** *est un donné par un* langage *(les énoncés que l'on peut formuler), un ensemble d'*axiomes *(énoncés de base qui sont admis) et un ensemble de* règles d'inférence ou règles de déduction *(qui nous permettent, à partir des axiomes, de démontrer de nouveaux énoncés).*

**Définition 9.** *Dans un système formel $F$ la notion de* **vérité** *est donnée par une* sémantique, *qui associe une valeur booléenne à toute formule close (souvent en utilisant la théorie des ensembles : vrai si et seulement si son ensemble de modèles est non-vide). Nous nous en tiendrons à l'idée intuitive de « vérité », et au fait que tout énoncé est soit vrai soit faux.*

**Définition 10.** *Dans un système formel $F$, une* **preuve** *d'un énoncé peut être vue comme un* arbre *dont la* racine *est l'énoncé prouvé, les* feuilles *sont des axiomes, et les* noeuds internes *correspondent à des applications des règles de déduction.*

Exemples de systèmes formels :
— Géométrie Euclidienne (https://en.wikipedia.org/wiki/Euclidean_geometry#Axioms).
— Artithmétique de Peano (https://fr.wikipedia.org/wiki/Axiomes_de_Peano).
— ZF et ZFC (https://en.wikipedia.org/wiki/Zermelo-Fraenkel_set_theory).
— Quel système formel contient toutes « les mathématiques » ? Réponse courte : ZF. Réponse longue : (https://en.wikipedia.org/wiki/Foundations_of_mathematics).

**Remarque 11.** *Les ensembles d'axiomes et de règles de déduction peuvent être infinis, mais il doivent être semi-décidables (***récursivement énumérable***).*

Voici maintenant les propriétés des systèmes formels que nous allons étudier.

**Définition 12.** *Un système formel est* **cohérent** *si l'on ne peut pas démontrer un énoncé et sa négation (*càd non-contradictoire*) (sinon on peut démontrer* tous *les énoncés).*

**Définition 13.** *Un système formel est* **complet** *si l'on peut démontrer tous les énoncés qui sont vrais (*càd *pour tout énoncé on peut démontrer soit l'énoncé soit sa négation).*

On peut alors formuler les théorèmes d'incomplétude de Gödel (1931) [6].

**Théorème 14** (incomplétude I)**.** *Aucun système formel contenant l'arithmétique élémentaire ne peut pas être à la fois cohérent et complet.*

**Théorème 15** (incomplétude II)**.** *Tout système formel cohérent et contenant l'arithmétique élémentaire ne peut pas démontrer sa propre cohérence.*

Donc même si on ne peut pas le démontrer (theorème 15), on *admet* comme hypothèse implicite à notre utilisation des mathématiques, qu'elles sont cohérentes (non-contradictoires). Il s'ensuit (théorème 14) que c'est un système formel incomplet (*e.g.* l'**hypothèse du continu** est indépendante de ZFC).

Les résultats de Gödel indiquent en particulier que les notions de vérité et de démonstrabilité diffèrent, car on ne peut pas démontrer tout ce qui est vrai. En revanche, un système formel qui ne démontre que des énoncés vrais (ce que l'on espère) est appelé **correct**.

## 2.3 Premier théorème d'incomplétude de Gödel

Avec l'accès à son propre code, on peut redémontrer le premier théorème d'incomplétude de Gödel assez simplement. Soit le programme suivant :

1. obtient son propre code,
2. énumère les preuves de ZFC [7] *jusqu'à* :
   — rencontrer une preuve de sa propre terminaison, et alors *boucler*,
   — rencontrer une preuve de sa propre non-terminaison, et alors *s'arrêter*.

Alors, est-ce que ce programme termine ? Il y a trois possibilités.

— Arrêt à l'instruction « *s'arrêter* », alors il existe une preuve de sa non-terminaison (pour arriver à cette partie là du code), et il existe aussi une preuve de sa terminaison (la liste des étapes de calcul jusqu'à l'arrêt), donc ZFC est **incohérente**.

— Non-arrêt à l'instruction « *boucler* », alors il existe une preuve de sa terminaison (pour arriver à cette partir là du code), et il existe aussi une preuve de sa non-terminaison (la liste des étapes de calcul menant à l'instruction *boucler*), donc ZFC est **incohérente**.

— Non-arrêt dans l'énumeration « *jusqu'à* », alors ZFC est **incomplète**.

En analysant ce programme, qui probablement ne s'arrête pas [10], on montre donc :

**Théorème 16.** *Aucun système formel permettant d'exprimer la notion de calcul et d'arrêt, ne peut être à la fois cohérent et complet.*

## 2.4 Sur la longueur des preuves

Source : [8].

On peut également démontrer très simplement que la longueur des preuves croît (en fonction de la longueur des énoncés) plus rapidement que toute fonction calculable (Gödel avait déjà observé cela [3]).

6. La formulation originale requiert des définitions plus subtiles ($\omega$-cohérence).

7. Ou tout autre système formel permettant d'exprimer la notion de calcul et d'arrêt, par exemple l'arithmétique de Peano est suffisante.

Soit $F$ un système formel **indécidable** et avec un ensemble d'axiomes et règles récursivement énumérables. Pour un énoncé $\phi$, on notera $L(\phi)$ la longueur de la plus courte preuve de $\phi$ si une telle preuve existe, et $L(\phi) = 0$ sinon. Soit $L(n)$ la valeur maximale de $L(\phi)$ pour les énoncés $\phi$ de longueur au plus $n$.

**Théorème 17.** *$L$ croît plus rapidement que toute fonction calculable.*

*Démonstration.* Par l'absurde, supposons qu'il existe une fontion calculable $f$ qui borne supérieurement $L$. Alors on peut décider $F$ : étant donnée une formule $\phi$ à décider, on a l'algorithme suivant :

1. calculer $f(|\phi|)$,
2. énumérer toutes les preuves de longueur au plus $f(|\phi|)$, et pour chacune d'elle vérifier si c'est une preuve de $\phi$,
3. si on trouve une preuve de $\phi$ alors accepter, sinon rejeter.

Cet algorithme termine puisqu'on ne vérifie qu'un ensemble fini de preuves (toutes celles de taille au plus $f(|\phi|)$), et est correct car si une preuve de $\phi$ existe, elle est par notre hypothèse de taille au plus $L(|\phi|) \leq f(|\phi|)$ donc on doit la rencontrer. Sinon c'est que $L(|\phi|) = 0$. Or notre système formel est indécidable, une contradiction. □

# Références


[1] S. Aaronson. Blog post : Rosser's Theorem via Turing machines. *https://www.scottaaronson.com/blog/?p=710*, 2011 (consulté en février 2019).

[2] A. K. Doxiàdis, C. Papadimitriou, A. Papadatos, and A. Di Donna. *Logicomix*. Vuibert (French edition), 2010.

[3] K. Gödel. On the length of proofs. *Traduction anglaise dans* The Undecidable : Basic Papers on Undecidable Propositions, Unsolvable Problems and Computable Functions*, edité par M. Davis*, 2004.

[4] D. Hofstadter. *Gödel, Escher, Bach : Les Brins d'une Guirlande Éternelle*. Dunod (French edition), 1979 (1989).

[5] S. Kritchman and R. Raz. The surprise examination paradox and the second incompleteness theorem. *Notices of the AMS*, 57(11), 2010.

[6] D. Madore. The fixed-point theorem. *http://www.madore.org/~david/computers/quine.html#sec_fp*, (consulté en février 2019).

[7] E. Nagel, J. .R. Newman, K. Gödel, and J.-Y. Girard. *Le théorème de Gödel*. Éditions du Seuil, 1989.

[8] A. E. Porreca. On the length of proofs (episode II). *https://aeporreca.org/blog/length-of-proofs-2*, 2010 (consulté en février 2019).

[9] M. Sipser. *Introduction to the theory of computation*. Course Technology, 2006.

[10] A. Yedidia and S. Aaronson. A relatively small turing machine whose behavior is independent of set theory. 2016. arXiv:1605.04343.