# Calculabilité
# Cours 5 : calculabilité avancée

Kévin Perrot – Aix Marseille Université – printemps 2019

## Table des matières



# 6 Kleene et Quine

[4] 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)

$$\phi_u(n, \dots) = \phi_n(\dots),$$

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 (les $\ll \dots \gg$).

Le **théorème s-m-n** est vrai : si une fonction $\phi_m$ est calculable, alors pour toute entrée $n$, il existe une fonction calculable $s(m,n)$ telle que

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

Ce théorème dit que l'on peut bêtement [1] coder en dur les arguments.

## 6.1 Théorème du point fixe de Kleene

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

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

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.* Pour un programme $t$, considérons le programme $s(t,t)$ donné par le théorème s-m-n, qui réalise ce que $t$ effectue s'il prend en entrée son propre code ou sa propre description. Considérons maintenant $h(s(t,t))$. Puisqu'il existe un interpréteur (une MT universelle) $\phi_u$ qui comprend le code qui lui est donné en entrée, il existe également une fonction $\phi_m(t,\dots) = \phi_{h(s(t,t))}(\dots)$ qui comprend l'entrée $t$, calcule le programme $h(s(t,t))$ et simule ce dernier sur le reste de l'entrée.

Alors nous pouvons affirmer que le programme $n = s(m,m)$ est le point fixe recherché. En effet, $\phi_n(\dots) = \phi_{s(m,m)}(\dots)$, et par définition de $s$, ceci est égal à $\phi_m(m,\dots)$, qui par définition de $m$, est $\phi_{h(s(m,m))}(\dots) = \phi_{h(n)}(\dots)$. □

Pour résumer cette preuve, nous avons pris le programme $m$ qui, étant donné un programme $t$, interprète le programme résultant de l'application de la transformation $h$ sur $t$ agissant sur lui-même, et nous avons appliqué ce programme à lui même.

## 6.2 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 $n$. □

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

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.

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. 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.

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);}
```

```
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)";;
```

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

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

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

## 6.3 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 $n$, il existe $self(n)$ tel que $\phi_{self(n)}(\dots) = \phi_n(n, \dots)$.*

*Démonstration.* Soit $n$ un programme qui suppose qu'il a accès à son propre code. Pour un programme $t$, considérons le programme $h(t)$ qui est $n$ dans lequel on a codé en dur le code de $n$ (dans $h(t)$ on oublie complètement le programme $t$). La fonction $h$ est calculable[5]. Appliquons le théorème 1 à la fonction $h$ : il existe la machine $self(n)$ qui est équivalent au programme $n$ ayant accès à son propre code codé en dur. □

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 est sensé décider 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.

**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 $x$, si le calcule de $f$ sur $x$ s'arrête, alors on peut construire le programme $g$ qui, sur une entrée $x$ :

1. obtient son propre code $g$,
2. calcule $\phi_h(g, x)$ (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 $x$ le résultat de $\phi_h(g, x)$ est incorrect, une contradiction. □

5. on peut écrire un programme $h$ qui prend en entrée le code de $t$, et le remplace par le code du programme $n$ dans lequel $n$ est codé en dur.

**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*[6] *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 $g$. □

# 7 Castor affairé (busy beaver)

Voir la planche de TD sur le sujet.

# 8 Calculabilité des nombres réels

Voir la planche de TD sur le sujet.

# 9 Arrêt et conjecture de Collatz

Si l'on savait décider le problème de l'arrêt (c'est-à-dire le résoudre avec une procédure algorithmique), alors on pourrait décider des conjectures mathématiques, comme par exemple la conjecture de Collatz.

Supposons donc qu'il existe un algorithme $H$ pour calculer la fonction d'arrêt *halt*, c'est-à-dire $H(A, w) = 1$ si le calcul de l'algorithme $A$ sur l'entrée $w$ s'arrête, et 0 sinon. Et considérons les algorithmes suivants.

```
collatz(n:int)
    si n == 1 alors stop, sinon
        si n%2 == 0 alors collatz(n/2), sinon collatz(3*n+1), finsi
    finsi

collatz_conjecture(n:int)
    si H(collatz,n)==1 alors collatz_conjecture(n+1), finsi
    stop
```

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

Alors l'algorithme `collatz_conjecture`, lancé sur 1, s'arrête si et seulement si il existe un contre exemple à la conjecture de Collatz ! Autrement dit, $H(\texttt{collatz\_conjecture}, 1)$ indique si la conjecture de Collatz est vraie (0) ou fausse (1).

# 10 Complexité de Kolmogorov

Voir [6] chapitre 6 section 4.

# 11 Et si nous avions la réponse au problème de l'arrêt ?

Imaginons (qu'il soit bien clair que cette section est purement conceptuelle) un instant que nous soit donnée une machine **oracle** qui résolve le problème de l'arrêt. Nous ne savons pas comment elle fonctionne, mais nous pouvons l'appeler autant de fois que nous voulons pour obtenir la réponse à des instances du problème de l'arrêt (étant données une machine de Turing $M$ et une entrée $w$, cet oracle nous répondra si le calcul $M(w)$ termine ou non). Sans cet oracle, nous pouvons calculer un certain sous-ensemble (très petit) de fonctions. Qu'en est-il si nous avons accès à cet oracle ? Nous pouvons bien calculer plus de fonctions, par exemple la fonction d'arrêt des machines de Turing était auparavant non calculable, mais elle est calculable si l'on a accès à cet oracle. Cependant, le problème de l'arrêt des *machines de Turing qui ont accès à l'oracle sur les machines de Turing* n'est pas calculable ! La preuve est identique à celle donnée plus tôt dans ce cours.

Imaginons alors que nous soit donnée une machine oracle qui résolve le problème de l'arrêt des *machines de Turing qui ont accès à l'oracle sur les machines de Turing*. Nous pouvons à nouveau calculer un plus grand ensemble de fonctions, mais le problème de l'arrêt des *machines de Turing qui ont accès à l'oracle sur les machines de Turing qui ont accès à l'oracle sur les machines de Turing* n'est pas calculable !

A chaque ajout d'un oracle qui résout le problème de l'arrêt pour un modèle donné, on fait ce qui s'appelle un **saut** (**Turing-jump**). En effectuant de tels sauts, nous nous baladons dans la hiérarchie des **degrés Turing**[7]... Nous pouvons faire 1 saut, 2 sauts, $\aleph_0$ sauts et même davantage ! Et il y aura toujours des fonctions non calculables.

Par exemple, le langage $L_\uparrow = \{\langle M\rangle \mid M(w) \uparrow \text{ pour tout } w\}$ a un degré Turing plus élevé que $L_{halt\epsilon} = \{\langle M\rangle \mid M \text{ s'arrête sur l'entrée } \epsilon\}$, qui lui même a le même degré Turing que $L_u = \{\langle M\rangle \# w \mid M \text{ s'arrête sur l'entrée } w\}$.

# 12 Premier théorème d'incomplétude de Gödel

[1] 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.

Un système formel est un langage (les énoncés que l'on peut formuler), un ensemble d'axiomes (énoncés de base dont la vérité est admise) et un ensemble de règles d'inférence ou règles de déduction (qui nous permettent de démontrer que de nouveaux énoncés

7. formellement, les degrés Turing sont les classes d'équivalence de langages pour les réductions avec oracle : deux langages $A$ et $B$ sont de même degré ssi $A \leq^T B$ et $B \leq^T A$.

mathématiques sont vrais). La notion de *vérité* est plus précisément donnée par une *sémantique*[8], mais nous allons ici nous concentrer sur la notion de *preuve* d'un énoncé, qui 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. Intuitivement, quand on dit « les mathématiques », on parle de la *théorie des ensembles de Zermelo–Fraenkel* (ZF), qui est un système formel. ZF inclue notamment l'arithmétique élémentaire (qui est définie, elle seule, par les axiomes de Peano).

Nous avons besoin de quelques définitions.

**Définition 8.** *Un système formel est* correct *si tous les énoncés que l'on peut prouver son vrais.*

La notion de correction indique que les règles d'un système formel mettent en oeuvre des raisonnement qui ont du sens : à partir d'énoncés vrais, les règles de déduction nous permettent uniquement de prouver (synonymes : dériver, déduire) des énoncés vrais.

**Définition 9.** *Un système formel est* complet *si l'on peut prouver tous les énoncés qui sont vrais.*

La complétude est une notion duale de la correction : si un énoncé est vrai, alors on peut le démontrer.

**Définition 10.** *Un système formel est* cohérent *si l'on ne peut pas démontrer un énoncé et sa négation.*

La cohérence indique que le système formel est *non-contradictoire*, ce dont on a bien sûr envie. On peut alors formuler le premier théorème d'incomplétude de Gödel (1931)[9].

**Théorème 11.** *Tout système formel cohérent et contenant l'arithmétique élémentaire est incomplet (on peut y construire un énoncé qui ne peut être ni prouvé ni réfuté).*

## 12.1 L'énoncé de Gödel

Pour démontrer le théoreme 11, Gödel utilise l'arithmétique du système formel $F$ pour encoder la notion de preuve dans les énoncés[10], et construire l'énoncé $G(F)$ suivant :

« Cet énoncé n'est pas prouvable dans $F$. »

Qui ressemble beaucoup au paradoxe du menteur qui affirme « cette phrase est fausse » : si la phrase est fausse c'est qu'elle est vraie, mais si elle vraie c'est qu'elle est fausse ! On remarquera également l'auto-référence, qui est un élément central dans ces énoncés.

8. une interprétation de la vérité des formules, souvent basée sur la théorie des ensembles, une formule close étant vraie si son interprétation est un ensemble non-vide.

9. ce n'est pas la formulation originale, qui est moins forte et requiert des définitions plus subtiles.

10. on appelle cela *l'arithmétisation des métamathématiques*. Le numéro de Gödel d'un énoncé $\phi = s_1 s_2 \dots s_n$ avec $s_i$ des symboles est donné par $g(\phi) = \Pi_{i=1}^{n} p_i^{g(s_i)}$ où $p_i$ est le $i^{\text{e}}$ nombre premier (l'ensemble des nombres premiers est r.e., on peut même les énumérer dans l'ordre) et $g(s_i)$ est le numéro du symbole $s_i$ (c'est une définition inductive, avec pour cas de base les symboles que l'on numérote). On décode $g(\phi)$ en calculant sa décomposition en facteur premiers, dont les exposants nous donnent les symboles de l'énonce. Les preuves sont des suites de symboles, elle ont aussi un numéro de Gödel. C'est un encodage (plutôt « mathématique » que « informatique ») des énoncés et preuves en des nombres, comme peut l'être l'encodage en binaire (mais les notions d'ordinateur et d'information allaient révolutionner notre compréhension des mathématiques et du monde quelques années plus tard...).

Attention, l'énoncé $G(F)$ montre en fait un résultat un peu moins fort que le théorème 11, dans lequel la notion de correction se substitue à la cohérence (la correction implique la cohérence, mais pas réciproquement [11]) :

- si $F$ prouve $G(F)$ alors $F$ n'est pas cohérent (on peut prouver $G(F)$ et $\neg G(F)$) et donc pas correct,
- si $F$ prouve $\neg G(F)$ alors
  - soit il existe une preuve de $G(F)$ (ce qu'affirme $\neg G(F)$) et alors $F$ n'est pas cohérent et donc pas correct,
  - soit il n'existe pas de preuve de $G(F)$ et alors $F$ n'est pas correct (puisque $\neg G(F)$ nous dit qu'une telle preuve existe).

On peut également démontrer cette version faible du théorème 11, très simplement, en se servant du théorème de l'arrêt !

*Démonstration du théorème 11 (version faible).* Par l'absurde, supposons que l'on ait un système formel $F$ correct et complet, qui soit suffisamment expressif pour raisonner sur les machines de Turing (c'est là que l'arithmétique élémentaire intervient). Alors on pourrait utiliser ce système formel pour résoudre le problème de l'arrêt : étant donnée $\langle M \rangle$ dont on se demande si elle s'arrête sur l'entrée vide, on a un algorithme qui consiste à énumérer toutes les preuves [12] du système $F$, jusqu'à rencontrer :

- ou bien une preuve que la machine $M$ s'arrête sur l'entrée vide,
- ou bien une preuve que la machine $M$ ne s'arrête pas sur l'entrée vide.

Puisque $F$ est complète, une de ces deux preuves existe et sera énumérée donc notre algorithme termine, et puisque $F$ est correcte la conclusion de cette preuve sera vraie. Donc on pourrait décider si $M$ s'arrête ou non, ce qui est en contradiction avec l'indécidabilité du problème de l'arrêt des machines de Turing. □

Ici on voit que $F$ correct plus complet implique $F$ décidable : on peut construire un algorithme (c'est ce que fait la preuve) qui, étant donné un énoncé, **décide** s'il admet une preuve ou si sa négation admet une preuve. Alors si le système formel $F$ est suffisamment expressif pour construire des énoncés qui parlent du comportement des machines de Turing (ce que permet tout système qui contient l'arithmétique élémentaire), on pourrait décider le problème de l'arrêt. Or on sait que ce n'est pas possible, contradiction.

## 12.2 L'énoncé de Rosser

Pour démontrer le théorème 11, il faut faire appel à l'idée de Rosser (1936) et construire l'énoncé $R(F)$ suivant (une *réfutation* d'un énoncé est une preuve de sa négation) :

« Pour toute preuve de cet énoncé dans $F$, il existe une réfutation plus courte. »

On a alors un raisonnement qui mène au théorème 11 :

- si $F$ prouve $R(F)$, alors cela prouve qu'il existe une réfutation de $R(F)$ plus courte que cette preuve, que l'on peut donc effectivement chercher (l'espace de recherche étant fini) et :

11. pour reprendre Scott Aaronson (*sound* signifie correct, et *consistent* signifie cohérent) : « If I believe that there's a giant purple boogeyman on the moon, then presumably my belief is *unsound*, but it might be perfectly *consistent* with my various other beliefs about boogeymen ».

12. si on imagine un ensemble fini d'axiomes et de règles de déduction, alors l'ensemble des preuves (arbres) que l'on peut construire est infini dénombrable et on peut les énumérer. On peut également énumérer toutes les preuves si les axiomes et règles sont récursivement énumérables (*e.g.* ZF).

— si on trouve une preuve de $\neg R(F)$ alors $F$ est incohérent,
— si on ne trouve pas une preuve de $\neg R(F)$, alors on vient de prouver $\neg R(F)$ (il n'existe pas de réfutation plus courte) et donc $F$ est incohérent,

— si $F$ prouve $\neg R(F)$ alors cela prouve qu'il existe une preuve de $R(F)$ plus courte que toute réfutation de $R(F)$, donc il existe une preuve de $R(F)$ plus courte que cette réfutation, que l'on peut donc effectivement chercher (l'espace de recherche étant fini) et :
— si on trouve une preuve de $R(F)$ alors $F$ est incohérent,
— si on ne trouve pas une preuve de $R(F)$, alors on vient de prouver $R(F)$ (il existe une réfutation plus courte que toute preuve) et donc $F$ est incohérent,

On remarquera la symétrie de l'argumentation obtenue grâce à l'énoncé de Rosser.

Pour relier l'énoncé de Rosser aux machines de Turing, on peut définir le **problème de devinette cohérente** suivant : étant donné le code $\langle M \rangle$ d'une machine de Turing, on cherche un algorithme (une machine de Turing) qui :
— si $M$ accepte $\epsilon$ alors accepte,
— si $M$ rejette $\epsilon$ en s'arrêtant alors rejette,
— si $M$ ne s'arrête pas sur $\epsilon$ alors accepte ou rejette, mais s'arrête !

On voit qu'il existe une symétrie dans ce problème entre l'arrêt acceptant et l'arrêt rejetant, avec le cas où $M$ ne s'arrête pas qui est en quelque sorte ignoré.

**Théorème 12.** *Le problème de devinette cohérente n'est pas décidable.*

*Démonstration.* Supposons qu'il existe une machine $P$ pour le résoudre, alors on peut construire la machine $Q$ qui, sur l'entrée $\langle M \rangle$,
— rejette si $M(\langle M \rangle)$ accepte,
— accepte si $M(\langle M \rangle)$ rejette,
— s'arrête (et accepte ou rejette) si $M(\langle M \rangle)$ ne s'arrête pas.

Que vaut $Q(\langle Q \rangle)$ ? Que le calcul accepte, rejette en s'arrêtant, ou ne s'arrête pas, on obtient une contradiction. □

*Démonstration du théorème 11.* Par l'absurde, supposons que l'on ait un système formel $F$ cohérent et complet (mais pas nécessairement correct !), qui soit suffisamment expressif pour raisonner sur les machines de Turing (c'est là que l'arithmétique élémentaire intervient). Alors on pourrait utiliser ce système formel pour résoudre le problème de devinette cohérente : étant donnée $\langle M \rangle$, on a un algorithme qui consiste à énumérer en parallèle toutes les possibles preuves et réfutations de l'énoncé « M accepte $\epsilon$ » dans le système $F$, jusqu'à rencontrer :
— une preuve de « M accepte $\epsilon$ », auquel cas on accepte,
— une réfutation de « M accepte $\epsilon$ », auquel cas on rejette.

Alors cet algorithme résout bien le problème de devinette cohérente. Tout d'abord, puisque $F$ est complète, une de ces deux preuves existe et sera énumérée donc notre algorithme termine. Ensuite, puisque $F$ est cohérente, cet algorithme ne fait pas d'erreur :
— si $M$ accepte vraiment $\epsilon$ alors $F$ ne peut pas prouver que $M$ n'accepte pas $\epsilon$, car sinon on pourrait se rendre compte que $M$ accepte $\epsilon$, et donc de l'incohérence de $F$, en temps fini,
— si $M$ rejette vraiment $\epsilon$ en s'arrêtant alors $F$ ne peut pas prouver que $M$ accepte $\epsilon$, car sinon on pourrait également se rendre compte que $M$ rejette $\epsilon$ en s'arrêtant, et donc de l'incohérence de $F$, en temps fini.

Cet algorithme résout donc bien le problème de devinette cohérente (s'arrête toujours, si $M$ accepte $\epsilon$ alors accepte, et si $M$ rejette $\epsilon$ en s'arrêtant alors rejette), or ce problème est indécidable (théorème 12), une contradiction. □

## 12.3 Sur la longueur des preuves

[5] On peut également démontrer très simplement que la longueur des preuves croit (en fonction de la longueur des énoncés) plus rapidement que toute fonction calculable (Gödel avait remarqué cela [2]).

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 13.** *$L$ 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 répondre *oui*, sinon répondre *non*.

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 de taille au plus $L(|\phi|) \leq f(|\phi|)$ donc on doit la rencontrer. Sinon c'est que $L(|\phi|) = 0$. □

## 12.4 Second théorème d'incomplétude de Gödel

**Théorème 14.** *Tout système formel cohérent et contenant l'arithmétique élémentaire ne peut pas démontrer sa propre cohérence.*

Voir [3].

## 12.5 Correspondance de Curry-Howard

Pour aller plus loin, il existe des liens très forts entre les notions de :
— preuve dans un système formel,
— programme dans un modèle de calcul.
C'est un peu comme si quand vous écrivez des programmes, vous écrivez des preuves... en fait, c'est même exactement cela !

# 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] 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.

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

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

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

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