Cours Logique et Calculabilité

L3 Informatique 2016/2017

Texte par Kévin Perrot
Version du 6 avril 2017

Table des matières

Chapitre 3

Calculabilité

3.0 Divertissement

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

Est-ce que le programme `collatz` termine sur toute entrée `n` ? Comment le savoir ?
— Tester pour tout $n \leq 10^{20}$?
— Tester si $\exists$ `n,x` tel que `collatz(n)=collatz`$^{\texttt{x}}$`(n)` ?
— Vous avez un an avec 10 programmeurs surdoués. Comment faites-vous ?

Suite de Collatz (1937) : $n \mapsto \begin{cases} n/2 & \text{si } n \mod 2 == 0 \\ n*3+1 & \text{sinon} \end{cases}$

Conjectures réfutées par de grands contre-exemples :
— Conjecture d'Euler (1772). $\forall n > 2 : \forall x_1, ..., x_k, z \in \mathbb{N} : \sum_{i=1}^{k} x_i^n \neq z^n$.
$n = 5$ (1966) : $27^5 + 84^5 + 110^5 + 133^5 = 144^5$
$n = 4$ (1988) : $2682440^4 + 15365639^4 + 18796769^4 = 20615673^4$
$n > 5$: inconnu.
— Conjecture de Pólya (1919). Plus de la moitié des entiers naturels inférieurs à un entier donné ont un nombre impair de facteurs premiers.
(1980) : le plus petit contre exemple est 906 150 257.
— Conjecture de Mertens (1885) : prouvée fausse mais aucun contre exemple explicite connu.
2006 : plus petit contre exemple compris entre 10^{14} et 1.59×10^{40}
— Nombre de Skewes (1914).
1955 : il existe un tel nombre inférieur à $10^{10^{10^{963}}}$.
1987 : il existe un tel nombre inférieur à 7×10^{370}.

La morale de cet example est que les ordinateurs n'ont pas réponse à tout !

Ce cours est basé sur le livre de Sipser [Sip06] et le cours de Kari [Kar13].

3.1 Introduction

Une machine de Turing est un objet mathématique, défini en 1936 par *Alan Turing*, qui a pour but de décrire ce qu'est un *calcul*. Tout le monde sent ce qu'est un calcul : si vous avez deux nombre en base 10, disons `123` et `456`, et vous voulez calculer leur somme, vous le faites chiffre par chiffre de droite à gauche en propageant d'éventuelles retenues pour obtenir `579`. Si vous voulez calculer leur produit, vous le décomposez en multiplications plus simples pour lesquelles vous avez appris un nombre fini de tables, et vous sommez les résultats des sous-problèmes :

```
123x456  =(1x100+2x10+3)x(4x100+5x10+6)
         =(1x4)x(100x100)+(1x5)x(100x10)+(1x6)x(100)
         +(2x4)x(10x100)+(2x5)x(10x10)+(2x6)x(10)
         +(3x4)x(100)+(3x5)x(10)+(3x6)
         =56088.
```

Lorsque vous apprenez à quelqu'un une méthode pour effectuer une multiplication, vous décrivez un *algorithme* : vous donnez une description finie (par exemple en 10 minutes de parole, ou en 2 pages) de la procédure à suivre pour obtenir le résultat. **Les machines de Turing sont des objets mathématiques pour décrire les algorithmes**. Une machine de Turing pour l'addition de nombres naturels en base 10 donne l'ensemble des instructions qu'il faut effectuer pour calculer la somme de deux nombres. Une remarque importante est que la description de la procédure est finie, mais elle permet de calculer la somme de n'importe quel couple de nombres : `123+456` ou `1234567890+2345678901` ou des nombres plus grands !

Bon, soyons sérieux. Une machine de Turing décrit comment calculer quelque chose. Mais quel est ce *quelque chose* ? C'est une *fonction*. Par exemple, une fonction peut-être l'addition, ou la multiplication : étant donnée une *entrée* finie (dans notre exemple `123` et `456`), une fonction associe une unique *sortie*. L'entrée et la sortie peuvent être un ou plusieurs entiers naturels, des nombres négatifs, une phrase écrite en alphabet Latin ou n'importe quel autre. Le point important est qu'elle soit de taille *finie*. Une fonction associe alors une unique sortie (le résultat) à chaque entrée.

Les fonctions peuvent être simples : l'addition ou la multiplication de deux entiers naturels ; ou plus compliquées : étant donné un entier naturel, quelle est sa décomposition en produit de facteurs premiers (entiers naturels plus grand que 1 qui n'ont pas de diviseur autre que 1 et lui-même) ; ou même *non calculable* : étant donné un énoncé mathématique, décider s'il est vrai ou faux. Oui, certaines fonctions ne sont pas calculables : il n'existe pas d'algorithme qui les calcul. De plus, il y a infiniment plus de fonctions non calculables que de fonctions calculables ! Il existe une infinité de fonctions, et une infinité de machines de Turing (d'algorithmes), mais le nombre de fonctions est infiniment plus grand que le nombre de machines de Turing.

Une question naturel est : si les machines de Turing peuvent calculer si peu de fonctions, pourquoi ne pas calculer avec un autre modèle mathématique ? En réalité, les machines de Turing ne sont pas le seul objet mathématique permettant de décrire des algorithmes. De tels modèles sont appelés *modèles de calcul effectifs*, où *effectif* signifie approximativement « en accord avec le monde réel ». Il est cependant magnifique de constater que tous les modèles de calcul proposés jusqu'à présent sont équivalents ! Deux modèles sont équivalents s'ils peuvent calculer exactement le même ensemble de fonctions. Rappelons nous bien : soit `F` l'ensemble de toutes les fonctions, les machines de Turing ne peuvent pas calculer toutes les fonctions de l'ensemble `F`, mais seulement un sous-ensemble `C`. **La croyance (répandue) selon laquelle tout autre modèle de calcul effectif que l'on pourrait imaginer sera à également capable de calculer toutes les fonctions de l'ensemble `C` et aucune autre est appelée *thèse de Church-Turing*, et `C` est appelé l'ensemble des *fonctions calculables***. L'une des questions les plus fondamentales de la science informatique est la suivante : pourquoi une fonction est-elle calculable ou non calculable ?

Un point intéressant est l'existence de machines de Turing *universelles*. Une machine de Turing universelle `U` est une machine capable de *simuler* tout autre machine de Turing. Qu'est-ce que cela signifie ? Soit `M` une machine de Turing quelconque et `x` une entrée, la sortie de `M` sur l'entrée `x` est

notée `M(x)`. Une machine `U` est universelle si l'on peut écrire une entrée `y` sur le ruban telle que le calcul de `U` sur `y` donne `M(x)` en sortie.

`U` et `y` ne sont pas très compliqués à construire : `M` a une description finie (principalement sa table d'actions), donc cette description peut être écrite sur le ruban, elle utilisera `n` cellules ; et sur d'autres cellules vides, on peut écrire `x`. Maintenant, `U` a toute l'information qui définit `M(x)` sur le ruban, et il est possible [1] de construire une telle machine `U` qui *lit* l'entrée `x` sur le ruban, ensuite *lit* la table d'action de `M` sur les `n` cellules dédiées du ruban, et ensuite réalise sur `x` ce que la machine `M` aurait réalisé si elle avait été exécutée sur un ruban contenant `x`. Une telle machine `U` est un peu délicate à construire, mais pas excessivement.

De nos jours, un ruban est appelé *disque dur*, la table d'action de `M` écrite sur le ruban est un *programme*, et `U` est un *ordinateur* !

Une machine de Turing universelle entièrement mécanique a été réalisée en Lego.

http://www.dailymotion.com/video/xrmfie/

http://rubens.ens-lyon.fr/

Par conséquent, ce tas de Lego est capable de calculer l'ensemble des fonctions calculables `C` : il a exactement la même *puissance de calcul* que votre ordinateur ou votre téléphone portable ! En comparaison avec un ordinateur moderne qui réalise une instruction toutes les nano-secondes (0.000000001 secondes), cette machine en Lego réalise une instruction toutes les 100 secondes. **Elle peut faire ce qu'un ordinateur moderne peut faire, mais pour réaliser ce que ce dernier effectue en 1 seconde, il lui faut 3168 ans 295 jours 9 heures 46 minutes et 40 secondes** [2]. Quoi qu'il en soit, l'important est qu'elle en soit capable, n'est-ce pas ?

C'est le coeur de la *calculabilité*. Les ordinateurs sont chaque année plus rapides, et leur vitesse continue d'augmenter. Cependant, ils restent restreints à l'ensemble des fonctions calculables, `C`. Ils peuvent calculer les fonctions de l'ensemble `C` toujours plus vite, mais ne peuvent pas s'échapper de `C` : leur *expressivité* reste la même. L'étude de cette expressivité, du sens de cette puissance de calcul, s'appelle la *théorie de la calculabilité*.

Vous pouvez parfois entendre que nous sommes aujourd'hui capables de calculer des choses qui étaient impossible à calculer les années passées, que les ordinateurs sont plus puissants aujourd'hui qu'hier. Ces phrases doivent être précisées. En réalité, ces calculs étaient simplement trop longs à réaliser les années passées (par exemple, il aurait fallut 100 ans si vous les aviez exécutés sur un ordinateur en l'an 2000), mais aujourd'hui vous pouvez les calculer en un temps raisonnable (par exemple 100 secondes) ce qui permet d'obtenir effectivement le résultat. Un exemple intéressant est le jeu des échecs : il est possible aujourd'hui, et il a toujours été possible, d'écrire un algorithme qui vous indique coup après coup la meilleure action possible, mais les ordinateurs actuels sont bien trop lents pour exécuter un tel algorithme jusqu'au bout... Néanmoins, un jour nous serons capables de réaliser ce calcul en un temps raisonnable, et ensuite jouer aux échecs avec un ordinateur deviendra définitivement ennuyant car nous serons sûrs et certains de perdre chaque partie [3]. Laissez moi répéter qu'un tel algorithme existe déjà et est facile à implémenter sur un ordinateur, les ordinateurs sont simplement trop lents pour réaliser les calculs en un temps raisonnable. La *théorie de la calculabilité* s'intéresse à des vérités mathématiques qui sont indépendantes du temps de calcul : la question n'est pas de savoir si quelque chose sera faisable dans 10 ou 20 ans, mais plutôt si quelque chose est fondamentalement faisable ou non, fondamentalement vrai ou faux.

3.2 Machines de Turing

Pour montrer qu'une fonction est calculable ou qu'un langage est décidable (distinction discutée en 3.2.2) il faut donner un algorithme. Pour montrer qu'une fonction n'est pas calculable ou qu'un

1. Remarquons que l'existence de fonctions non calculables implique que pour d'autres problèmes (encore une fois il y en a énormément, infiniement plus que des calculales), même avec toute l'information qui définit la question, il n'existe pas de machine de Turing qui calcule le résultat.

2. Ce nombre est en réalité complètement faux, parce qu'une instruction de machine de Turing est différente d'une instruction d'un ordinateur moderne. Néanmoins, il est là pour souligner le fait que cette machine de Turing en Lego est très précisément équivalente à un ordinateur moderne

3. Je mens un peu. En réalité, puisque nous n'avons encore jamais calculé toutes les actions possibles aux échecs, nous ne savons pas si ce sont les blancs ou les noirs qui ont une stratégie gagnante, ou si nous arriverions à un match nul avec deux joueurs parfaits...

langage est indécibable, il faut d'abord définir l'ensemble des algorithmes (car cela définit l'ensemble de ce qui est calculable/décibable). L'intérêt des machines de Turing est qu'elles définissent les algorithmes de façon intuitive et simple ! Imaginez devoir définir mathématiquement votre langage de programmation préféré dans ses moindres détails...

Idée du calculateur humain devant sa feuille :
— feuilles découpées en cases : ruban ;
— crayon posé sur une case : tête de lecture/écriture, déplacement ;
— l'opérateur dispose d'une mémoire finie (son cerveau) : états.

3.2.1 Définitions

Définition 3.1. Une **machine de Turing** (**MT**) déterministe est un 7-uplet

$$M = (Q, \Sigma, \Gamma, \delta, q_0, B, q_F)$$

où
— Q est un **ensemble d'états** fini ;
— Σ est l'**alphabet d'entrée** ;
— Γ est l'**alphabet de ruban**
(il contient tous les symboles qui peuvent apparaître sur le ruban. En particulier $\Sigma \subset \Gamma$ car l'entrée est initialement écrite sur le ruban. On supposera que $Q \cap \Gamma = \varnothing$ pour qu'il n'y ait pas de confusion entre états et symboles du ruban) ;
— δ est une **fonction de transition** décrite ci-après ;
— $q_0 \in Q$ est l'**état initial** ;
— $B \in \Gamma \setminus \Sigma$ est un **symbole blanc** spécial (ne fait pas partie de l'alphabet d'entrée) ;
— $q_F \in Q$ est l'état final.

La fonction de transition δ est une application (partielle)
de l'ensemble $(Q \setminus \{q_F\}) \times \Gamma$ dans l'ensemble $Q \times \Gamma \times \{L, R\}$.
L'application est partielle : elle peut être indéfinie pour certains arguments, auquel cas la machine n'a pas de mouvement suivant et s'arrête. En particulier, il n'y a pas de transition depuis l'état final q_F. Une transition

$$\delta(q, a) = (p, b, L)$$

signifie que dans l'état q et en lisant le symbole de ruban a, la machine passe dans l'état p, remplace a par b sur le ruban, et déplace la tête de lecture/écriture d'une cellule sur la gauche (L pour *left* et R pour *right*).

Initialement, le mot d'entrée est écrit sur le ruban et toutes les autres cellules contiennent le symbole blanc B. La machine est dans l'état q_0, et la tête de lecture/écriture est positionnée sur la lettre la plus à gauche de l'entrée. Il y a trois possibilités :

— **acceptation** si au cours des transitions la machine entre dans l'état final q_F (et donc s'arrête),
— **rejet** si au cours des transitions la machine s'arrête dans un état non final (s'il n'y a pas de mouvement suivant à réaliser),
— **rejet** si la machine ne s'arrête jamais.

Définition 3.2. Une **description instantanée** (**DI**) d'une MT décrit sa configuration courante. C'est un mot

$$uqav \in (\{\epsilon\} \cup (\Gamma \setminus \{B\})\Gamma^*)Q\Gamma(\{\epsilon\} \cup \Gamma^*(\Gamma \setminus \{B\})$$

avec $q \in Q$ l'état courant, $u, v \in \Gamma$ le contenu du ruban à gauche et à droite de la tête, respectivement, jusqu'au dernier symbole non blanc, et $a \in \Gamma$ le symbole de ruban actuellement sous la tête.

Définition 3.3. Un **mouvement**, une **transition**, un **déplacement** de la MT à partir de la DI $\alpha = uqav$ vers la DI suivante β sera noté $\alpha \vdash \beta$. Plus précisément :

1. Si $\delta(q, a) = (p, b, L)$,
   — si $u = \epsilon$ alors $\beta = pBbv$ (potentiellement en supprimant des B à la fin de bv),

— si $u = u'c$ avec $c \in \Gamma$ alors $\beta = u'pcbv$ (potentiellement en supprimant des B à la fin de bv).

2. Si $\delta(q, a) = (p, b, R)$,
   — si $v = \epsilon$ alors $\beta = ubpB$ (potentiellement en supprimant les B au début de ub),
   — si $v \neq \epsilon$ alors $\beta = ubpv$ (potentiellement en supprimant les B au début de ub).
3. Si $\delta(q, a)$ est indéfini alors aucun mouvement n'est possible depuis α, et α est une **DI d'arrêt**. Si $q = q_F$ alors α est une **DI acceptante**.

Notation 3.4. Notre modèle de MT est **déterministe**, ce qui signifie que pour tout α il y a au plus un β tel que $\alpha \vdash \beta$. Nous noterons

$$\alpha \vdash^* \beta$$

si la MT change α en β en n'importe quel nombre d'étapes (0 inclus, auquel cas $\alpha = \beta$),
$\alpha \vdash^+ \beta$ si la MT change α en β en au moins une étape, et
$\alpha \vdash^i \beta$ si la MT change α en β en exactement i étapes.

Pour tout $w \in \Sigma^*$ nous pouvons définir la DI correspondante

$$\iota_w = \begin{cases} q_0 w, & \text{si } w \neq \epsilon \\ q_0 B & \text{si } w = \epsilon. \end{cases}$$

3.2.2 Décider et calculer

Définition 3.5. Le langage **reconnu** (ou **accepté**) par la MT M est

$$L(M) = \{w \mid w \in \Sigma^* \text{ et } \iota_w \vdash^* uq_F v \text{ avec } u, v \in \Gamma^*\}$$

Définition 3.6. Un langage est **récursivement énumérable** (**r.e.**) s'il est reconnu par une machine de Turing. Un langage est **récursif**, ou **décibable**, s'il est reconnu par une machine de Turing qui s'arrête sur toute les entrées.

Attention à la différence ! Bien entendu, tout langage récursif est également r.e.

Notation 3.7. Le **résultat** du calcul de la MT M sur l'entrée w sera noté

$$M(w) = \begin{cases} uv & \text{si } \iota_w \vdash^* uq_F v \text{ avec } u, v \in \Gamma^* \\ uav & \text{si } \iota_w \vdash^* uqav \text{ avec } u, v \in \Gamma^* \text{ et } \delta(q, a) \text{ non défini}\dagger \\ \uparrow & \text{si l'exécution ne termine pas.} \end{cases}$$

†potentiellement en supprimant des B à la fin de av.

Définition 3.8. Une fonction $f : \Sigma^* \to \Gamma^*$ est **calculable** ou **récursive** si et seulement si il existe une MT M telle que pour tout $w \in \Sigma^* : f(w) = M(w)$.

Remarque 3.9. Décider (un langage) est équivalent à calculer (une fonction).
⇒ Décider un langage L revient à calculer sa **fonction caractéristique**

$$f_L : \begin{array}{rl} \Sigma^* & \to \{0, 1\} \\ w & \mapsto \begin{cases} 1 & \text{si } w \in L \\ 0 & \text{sinon.} \end{cases} \end{array}$$

⇐ Calculer une fonction f revient à décider le langage

$$\{(x, y) \mid y = f(x)\}$$

Remarque 3.10.
— Il existe une bijection entre Σ^* et $\mathbb{N}$
— Il existe une **bijection entre** $\mathbb{N} \times \mathbb{N}$ **et** $\mathbb{N}$,
et donc également entre $\mathbb{N}^n$ et $\mathbb{N}$ pour tout $n \in \mathbb{N}$.

Par conséquent, lorsque nous parlons de fonctions calculables, nous pouvons restreindre nos considérations aux **fonctions de $\mathbb{N}$ dans $\mathbb{N}$**.

Nous n'établirons pas de distinction très nette entre calculer et décider.
— Dans la vraie vie on dira plutôt calculer (plus parlant).
— Dans le monde des mathématiques on dira plutôt décider (plus minimaliste, et s'applique aux propriétés).
— Récursif est synonyme de calculable et décidable.

Remarque 3.11. Le terme **récursivement énumérable** (définition 3.6) vient du fait qu'il est possible d'écrire (pour ces langages) une MT qui va, à partir d'une entrée vide, énumérer tous les mots du langage, un à un et sans en oublier aucun (dans n'importe quel ordre, possiblement en répétant plusieurs fois certains mots). C'est-à-dire que nous aurons un *état spécial d'énumération* q_e (quand on entre dans cet état c'est qu'on énumère le mot présent sur le ruban, par convention à la droite de la tête de lecture/écriture) tel que :

$$\text{pour tout mot } w \in L, \text{ il existe une étape } t \text{ telle que } \epsilon \vdash^t q_e w.$$

Exemple 3.12. Le langage suivant est récursif :

$$\{w \in \{a,b\}^* \mid w \text{ est un palindrome}\}$$

donc il existe une machine $M_{palindrome}$ qui le décide (répond oui/non sur toute entrée).

3.2.3 Propriétés de clôture

Théorème 3.13. *Les propriétés suivantes sont vraies :*

1. *la famille des langages récursifs est close par complémentation ;*
2. *les familles des langages récursifs et récursivement énumérables sont closes par union et intersection ;*
3. *Un langage $L \subseteq \Sigma^*$ est récursif si et seulement si L et $\Sigma^* \setminus L$ sont récursivement énumérables.*

Idées de démonstration.

1. On veut prouver L récursif implique $\Sigma^* \setminus L$ récursif. Soit M la machine dont le langage est $L(M) = L$ et qui s'arrête toujours, nous allons construire une nouvelle machine M' dont le langage est $L(M') = \Sigma^* \setminus L$ et qui s'arrête toujours. Pour cela, on ajoute un puits global qui sera notre nouvel état final. Ainsi, la nouvelle machine s'arrête dans cet état final à chaque fois que M s'arrêtait sur un état non final ($w \notin L(M) \Rightarrow w \in L(M')$). De plus, chaque fois que M s'arrêtait dans l'état final, M' va dans un nouvel état non final à partir duquel aucune transition n'est possible ($w \in L(M) \Rightarrow w \notin L(M')$).
2. Intersection. Soient $L_1 = L(M_1)$ et $L_2 = L(M_2)$. On veut construire une machine M' dont le langage est $L(M') = L_1 \cap L_2$. Pour cela, sur une entrée w la machine M' simulera (pour cela il suffit de modifier l'état final des machines simulées) :
   — M_1 sur l'entrée w (on sait que le calcul termine),
   — puis M_2 sur l'entrée w (on sait que le calcul termine),
   se souviendra du résultat de chaque simulation (arrêt dans l'état final ou non), et va dans l'état final si et seulement si M_1 et M_2 acceptent w (sinon M' va dans un nouvel état non final à partir duquel aucune transition n'est possible). Pour les langages r.e. on étudie en plus les cas où les machines M_1 et M_2 ne s'arrête pas.
3. Le sens $\Rightarrow$ est direct. Pour $\Leftarrow$, on simule en parallèle les deux machines qui reconnaissent L et Σ^* (mais ne s'arrête pas toujours). Puisque soit l'une soit l'autre accepte w, soit l'une soit l'autre entrera dans son état final, et nous pourrons alors :
   — entrer dans notre état final si c'est la machine qui reconnait L qui est entrée dans son état final,
   — entrer dans un nouvel état non final à partir duquel aucune transition n'est possible si c'est la machine qui reconnait $\Sigma^* \setminus L$ qui est entrée dans son état final.

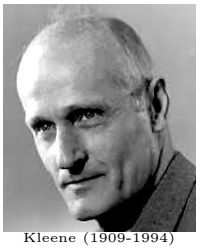
□

3.2.4 Un peu d'histoire


Cantor (1845-1918)

Hilbert (1862-1943)

Gödel (1906-1978)

Church (1903-1995)

Kleene (1909-1994)

Turing (1912-1954)

A la toute fin du XIX^{e} siècle, Georg Cantor définit les fondements de la théorie des ensembles, dont l'usage systématique (c'est-à-dire qui est utilisée dans tous les domaines) allait bouleverser les fondements de la logique mathématique. En 1900, pour fêter le passage au XX^{e} siècle, David Hilbert énonce 23 grands problèmes ouverts, dont le suivant : les propriétés qui s'expriment en langage mathématique sont-elles toutes décidables ? Si la réponse devait être affirmatives, les propriétés mathématiques valides seraient des théorèmes dérivables mécaniquement de quelques axiomes dans un système formel. Autrement dit : on pourrait remplacer les mathématicienNEs par des machines surpuissantes ! En 1931, Kurt Gödel met un terme à cette interrogation : il existe des propriétés mathématiques indécidables (dans tous les systèmes d'axiomes qui formalisent au moins l'arithmétique). Autrement dit : mathématicienNEs 1 - machines 0. Entre 1932 et 1936, Alsonso Church et Stephen Kleene proposent des modèles de calculs (le λ-calcul et les fonctions μ-récursives) qui semblent capturer la notion intuitive de fonctions calculables, mais il est un peu difficile de s'en convaincre... Notons tout de même que le λ-calcul est extrêmement minimaliste, ce qui rend sa compréhension mathématique fort intéressante : tout est capturé en quelques lignes de définition ! Indépendamment, en 1936, Alan Turing propose sa définition de machines. En 1937 il montre que la classe des fonctions λ-calculables est égale à la classe des fonctions programmables sur les machines de Turing. Les machines de Turing permettent de reformuler en termes intuitifs de calculs les résultats de Kurt Gödel (qui étaient exprimés en termes de démonstration). Avec l'aide de Von Neumann (et d'autres), les premiers ordinateurs programmables verront le jour quelques années plus tard !
La vie de Turing vaut le coup d'oeil (savez vous que le rôle de Turing durant la seconde guerre mondiale est resté secret d'Etat de nombreuses années ?).
e-penser (13') : https://www.youtube.com/watch?v=7dpFeXV_hqs

3.3 Les limites du calcul

3.3.1 Enumération des machines de Turing

Remarque 3.14.
Enumérer un ensemble S = donner (au moins) un numéro à chaque élément
= donner une fonction surjective de $\mathbb{N}$ dans S.
Dans ce cas S ne peut pas être plus grand que $\mathbb{N}$.
Une énumération [4] de S sans répétition (= injective) est une bijection de $\mathbb{N}$ dans S.

Remarque 3.15. Il y a

$$((nm2)+1)^{(n-1)m}$$

machines de Turing à $|Q| = n$ états et $|\Gamma| = m$ symboles. Il faut au moins 2 états (q_0 et q_F) et au moins 2 symboles de ruban (B et un autre) pour définir une MT.

Remarque 3.16. On peut écrire la définition d'une machine de Turing sur une feuille découpée en cases (ça ressemble beaucoup à un ruban). En prenant un ordre sur les symboles utilisés (ça ressemble beaucoup à un ordre sur Γ), on peut définir un ordre lexicographique [5] sur l'ensemble des machines de Turing. Donc étant donné un ensemble fini quelconque de machines de Turing, on peut les énumérer.

4. Il faut que l'énumération soit totale, c'est-à-dire que tout élément ait une image.
5. Comme dans un dictionnaire.

Lemme 3.17. *Il existe une bijection entre l'ensemble des machines de Turing et* $\mathbb{N}$.

Donc il existe une **énumération de l'ensemble de toutes les machines de Turing** (et même plusieurs).

Démonstration. Nous allons énumérer sans répétition l'ensemble des machines de Turing. L'idée est de numéroter de 0 à 80 les 81 MT à 2 états et 2 symboles, puis de 81 à 24277 les 2197 MT à 2 états et 3 symboles, puis de 24278 à 30838 les 28561 MT à 3 états et 2 symboles, puis de 30339 à 47076219 les 47045881 MT à 3 états et 3 symboles, etc.

Plus formellement, soit f une bijection de $\mathbb{N}$ dans $(\mathbb{N} \setminus \{0,1\})^2$. Nous allons commencer par énumérer les MT avec $(|Q|,|\Gamma|) = f(0)$, puis les MT avec $(|Q|,|\Gamma|) = f(1)$, etc. Pour un $(|Q|,|\Gamma|)$ donné il y a un nombre fini de machines de Turing (remarque 3.15), nous pouvons donc les énumérer sans problème (remarque 3.16). □

3.3.2 Simplifications

Lorsque cela nous arrangera, nous pourrons nous ramener à ne considérer
— que **les langages sur** $\Sigma = \{0,1\}$ (au lieu de Σ fini quelconque),
— que **les fonctions de** $\mathbb{N}$ **dans** $\mathbb{N}$ (au lieu des fonctions de Σ^* dans Γ^*).

Remarque 3.18. Quand on dit qu'une chose A « peut se ramener à » une chose B, on dit en fait plus formellement que l'ensemble des choses A peut être mis en **bijection** avec l'ensemble des choses B.

Pour un Σ donné, avec une bijection de Σ^* dans $\mathbb{N}$ (par exemple à partir de l'idée de compter en base $|\Sigma|$ on peut définir $f(w_0 w_1 \dots w_k) = \sum_{i=0}^{k} w_i m^i + m^{k+1} - 1$ avec $m = |\Sigma|$ et la convention $f(\epsilon) = 0$) on peut traduire un mot en un nombre (et réciproquement car c'est une bijection). En prenant la représentation binaire de ce nombre, nous obtenons une bijection de Σ^* dans $\{0,1\}^*$.

Avec une bijection de Σ^* dans $\mathbb{N}$ et une bijection de Γ^* dans $\mathbb{N}$, toute fonction de Σ^* dans Γ^* calculée par une MT peut se ramener à une fonction de $\mathbb{N}$ dans $\mathbb{N}$.

3.3.3 Dénombrement

Notation 3.19. L'**ensemble des parties de** $\mathbb{N}$ (ensemble des sous-ensembles de $\mathbb{N}$) est dénoté $\mathcal{P}(\mathbb{N})$ ou $2^{\mathbb{N}}$, et est en bijection avec $\mathbb{R}$, qui est en bijection avec $[0,1[$.

Notation 3.20. $|\mathbb{N}| = |\mathbb{Q}| = |\mathbb{Z}| = |\mathbb{N} \times \mathbb{N}| = \aleph_0$ et $|\mathbb{R}| = |\mathbb{R} \times \mathbb{R}| = |[0,1[| = 2^{\aleph_0}$.

Lemme 3.21. *Il existe une bijection entre l'ensemble des langages sur* $\Sigma = \{0,1\}$ *et* $[0,1[$.

Démonstration. L'ensemble des mots sur $\Sigma = \{0,1\}$ est en bijection avec $\mathbb{N}$ (représentation binaire). Un langage peut donc se ramener à un sous-ensemble de $\mathbb{N}$ (un ensemble de nombres). Par conséquent l'ensemble des langages sur $\{0,1\}$ est en bijection avec $\mathcal{P}(\mathbb{N})$, qui est en bijection avec $[0,1[$ (notation 3.19). □

Lemme 3.22. *Il existe une bijection entre l'ensemble des fonctions de* $\mathbb{N}$ *dans* $\mathbb{N}$, *et* $[0,1[$.

Démonstration. Il y a autant de fonctions de A dans B que d'éléments dans $B^{|A|}$. Dans notre cas il y a $\aleph_0^{\aleph_0}$ fonctions de $\mathbb{N}$ dans $\mathbb{N}$. Montrons que ce nombre n'est pas plus grand que $2^{\aleph_0}$, ce que nous admettrons comme suffisant pour conclure.

Pour cela nous allons construire une fonction injective de l'ensemble des fonctions de $\mathbb{N}$ dans $\mathbb{N}$, dans $[0,1[$, ce qui suffira pour conclure. Une fonction f de $\mathbb{N}$ dans $\mathbb{N}$ est une suite infinie de nombres : $f(0), f(1), f(2)$, etc (attention : chaque $f(i)$ est un nombre fini car $+\infty \notin \mathbb{N}$). Nous pouvons donc faire correspondre à chaque fonction un nombre réel $0.f(0)f(1)f(2)\dots$. Cette fonction n'est cependant pas injective (ni surjective). Pour la rendre injective nous pouvons utiliser le codage suivant :
— les $f(i)$ sont codés en binaire dédoublé (chaque bit est écrit deux fois, par exemple 9 en décimal devient 11000011),
— les $f(i)$ et $f(i+1)$ sont séparés par la séquence 01.

□

Théorème 3.23. $\aleph_0 < 2^{\aleph_0}$**.**

Démonstration. Nous allons démontrer ce résultat par l'absurde. Supposons qu'il existe une bijection entre $\mathbb{N}$ et $[0,1[$ (auquel cas $\aleph_0 = 2^{\aleph_0}$), alors il est possible d'énumérer tous les nombres réels entre 0 (inclus) et 1 (exclu) sans en oublier aucun :

$$\begin{aligned} r_1 &= 0.\boxed{1}234567890\ldots \\ r_2 &= 0.5\boxed{3}49236423\ldots \\ r_3 &= 0.72\boxed{9}1655000\ldots \\ r_4 &= 0.239\boxed{3}218693\ldots \\ &\ldots \end{aligned}$$

Nous allons montrer qu'il est impossible d'avoir énuméré tous les éléments de $[0,1[$. En effet, nous avons forcément oublié le nombre suivant :

$$r_+ = 0.2404\ldots$$

construit en prenant pour première décimale la première décimale de r_1 plus 1 modulo 10, en seconde décimale la seconde décimale de r_2 plus 1 modulo 10, en troisième décimale la troisième décimale de r_3 plus 1 modulo 10, etc, à l'infini (les nombres réels peuvent avoir une infinité de décimales). On a bien $r_+ \in [0,1[$, et pour tout $i \in \mathbb{N} : r_i \neq r_+$ car ils diffèrent par la $i^{\text{ème}}$ décimale. (Diagonale de Cantor, 1891) □

Corollaire 3.24.
Il existe des langages non récursifs et des fonctions non calculables.

Démonstration. Application du théorème 3.23 d'après les lemmes 3.17 et 3.21 pour les langages, et lemmes 3.17 et 3.22 pour les fonctions. □

Remarque 3.25. Il existe donc des infinis de tailles différentes. On dira
- **infini dénombrable** s'ils sont en bijection avec $\mathbb{N}$ (donc de taille $\aleph_0$),
- **infini indénombrable** sinon.

Le théorème 3.24 (corollaire du théorème 3.23) ne nous donne pas d'exemple de langage non récursif / fonction non calculable, mais nous dit qu'ils / elles sont très nombreux / nombreuses (infiniment plus que les récursifs / calculables).

Le théorème 3.23 amène une question naturelle : existe-t-il des infinis strictement plus grands que $\aleph_0$, mais strictement plus petits que $2^{\aleph_0}$? En d'autres termes, si l'on dénote $\aleph_1$ le second plus petit infini après $\aleph_0$, est-ce que

$$2^{\aleph_0} = \aleph_1 \,?$$

Cette question fameuse est appelée **hypothèse du continu** (HC), et fut posée par Cantor autour de 1878. Il fallut attendre l'axiomatisation de la théorie des ensembles [6] par Zermelo et Fraenkel (ZF) au début du XX$^{\text{e}}$ siècle, une preuve par Gödel en 1938 que HC ne peut pas être réfutée dans ZF, et une preuve par Cohen en 1963 que HC ne peut pas être prouvée dans ZF, pour arriver à la conclusion suivante : HC est indépendante de ZF. Reformulé, la théorie des ensembles qui fait consensus en mathématique ne permet pas de dire si l'hypothèse du continu est vraie ou fausse, les deux éventualités sont consistantes (n'amènent pas de contradiction).

6. C'est-à-dire la définition précise d'axiomes à partir desquels on dérive des théorèmes (vérités mathématiques). Avant cela (et pour Cantor notamment), on utilisait une définition intuitive (« naïve ») des ensembles. Par exemple, rien n'interdisait de considérer l'ensemble de tous les ensembles, $\mathcal{S}$. Russell souleva a ce propos un paradoxe divertissant : soit $X = \{A \in \mathcal{S} \mid A \notin A\}$, est-ce que $X \in X$?

3.3.4 Code d'une machine de Turing

Les résultats fondamentaux sur les limites du calcul sont liés à des problèmes dans lesquels une machine de Turing doit répondre à une question sur les machines de Turing. Pour cela, il faut pouvoir donner en entrée à une machine de Turing la définition (le code, le programme) d'une autre machine de Turing. Deux possibilités :

— en donnant le numéro de la machine dans une énumération des machines de Turing,
— en écrivant le code de la machine sur la ruban.

Nous avons vu dans la section 3.3.1 comment énumérer les machines de Turing, voyons maintenant comment les encoder sur le ruban.

Notation 3.26. Nous noterons $\langle M \rangle$ le **code d'une machine de Turing**.

Il y a de nombreuses façons d'encoder les machines de Turing sur le ruban. Par exemple, en numérotant de q_1 à q_n les états et de a_1 à a_m les symboles de ruban utilisés par une machine M, et en fixant $D = 0$ et $L = 00$, il est possible d'encoder chaque transition $\delta(q_i, a_j) = (q_k, a_l, D)$ de M par la séquence

$$\text{transition} = \underbrace{0\ldots0}_{i}1\underbrace{0\ldots0}_{j}1\underbrace{0\ldots0}_{k}1\underbrace{0\ldots0}_{l}10$$

On peut alors encoder une machine complète en commençant par dire combien elle a d'états, combien elle a de symboles de ruban, puis en listant les x transitions une à une :

$$\langle M \rangle = 111\underbrace{0\ldots0}_{n}11\underbrace{0\ldots0}_{m}11\text{transition}_1 11\text{transition}_2 11\ldots 11\text{transition}_x 111.$$

Par convention, nous pouvons énumérer les transitions dans l'ordre lexicographique selon l'état et le symbole.

On se convaincra que le résultat suivant est vrai.

Lemme 3.27. *Le langage* $L_{enc} = \{w \in \{0,1\} \mid w = \langle M \rangle$ *pour une MT* $M\}$ *est récursif.*

3.3.5 Théorème de l'arrêt

Théorème 3.28. *La fonction* **halt** : $(\langle M \rangle, w) \mapsto \begin{cases} 0 \text{ si } M(w) \uparrow \\ 1 \text{ sinon} \end{cases}$ *n'est pas calculable.*

Démonstration. Par l'absurde, supposons qu'il existe une machine de Turing M_{halt} qui calcule la fonction *halt*. Nous pouvons alors sans difficulté construire la machine M_{diag} suivante :

$$M_{diag}(i) = \begin{cases} 1 \text{ si } M_{halt}(\langle i \rangle, \langle i \rangle) = 0 \\ \uparrow \text{ si } M_{halt}(\langle i \rangle, \langle i \rangle) = 1 \end{cases}$$

où $\uparrow$ signifie que M_{diag} entre dans une boucle infinie (et ne termine donc pas). Considérons à présent l'entrée $\langle diag \rangle$ donnée à la machine M_{diag}. Deux cas sont possibles.

— Si $M_{diag}(\langle diag \rangle) = 1$ alors, par définition de M_{diag}, nous avons $M_{halt}(\langle diag \rangle, \langle diag \rangle) = 0$ ce qui signifie, par définition de M_{halt}, que $M_{diag}(\langle diag \rangle) \uparrow$, une contradiction.
— Si $M_{diag}(\langle diag \rangle) \uparrow$ alors, par définition de M_{diag}, nous avons $M_{halt}(\langle diag \rangle, \langle diag \rangle) = 1$ ce qui signifie, par définition de M_{halt}, que $M_{diag}(\langle diag \rangle)$ s'arrête, une contradiction.

Dans les deux cas nous arrivons à une contradiction. □

3.3.6 Raisonnement par l'absurde et diagonalisation

N'est-il pas surprenant que des résultats si révolutionnaires admettent des preuves si courtes et simples ? On peut noter le caractère diagonal (auto-référant) des preuves de Cantor (1891) et Turing (1936). C'est également sur un argument diagonal qu'est basé le premier théorème d'incomplétude de Gödel (1931) !

3.3.7 Réductions (version simple)

Une des formulations les plus populaires du résultat de Turing en 1936 est donnée par le théorème 3.28. Voyons maintenant des développements mathématiquement plus « épurés » qui mènent au même résultat, mais nous permettront d'aller plus loin de façon élégante [7].

Théorème 3.29. *Le langage* $L_d = \{\langle M\rangle \mid M \text{ n'accepte pas le mot } \langle M\rangle\}$ *n'est pas re.*

Démonstration. Par l'absurde, supposons qu'une machine M_d reconnaisse L_d. Considérons alors l'entrée $\langle M_d\rangle$ donnée à la machine M_d. Deux cas sont possibles.

- Si M_d n'accepte pas $\langle M_d\rangle$ alors, par définition du langage L_d, le mot $\langle M_d\rangle$ est dans L_d. Or M_d reconnait L_d, donc M_d accepte $\langle M_d\rangle$, une contradiction.
- Si M_d accepte $\langle M_d\rangle$ alors, par définition du langage L_d, le mot $\langle M_d\rangle$ n'est pas dans L_d. Or M_d reconnait L_d, donc M_d n'accepte pas $\langle M_d\rangle$, une contradiction.

Dans les deux cas nous arrivons à une contradiction. □

La preuve est cette fois encore plus simple ! Ce résultat nous dit qu'il n'existe pas de MT M_d (un seul et même algorithme qui répond oui/non correctement pour chaque instance) pour décider si une machine M reconnait le mot $\langle M\rangle$ (même si la machine M_d a le droit de ne pas s'arrêter si M ne reconnait pas $\langle M\rangle$). Ce problème peut sembler artificiel, mais il sert de *graine* pour dériver la non récursivité d'autres problèmes (plus naturels), via une méthode qui s'appelle une **réduction**.

On dira qu'un problème A se réduit à un problème B si connaissant un algorithme pour décider/calculer B, on peut obtenir un algorithme pour décider/calculer A.

Définition 3.30. Un problème de décision A est un ensemble d'instances I_A et une fonction $f_A : I_A \to \{0, 1\}$. Une **many-one-réduction Turing** de A à B est alors une fonction calculable $g : I_A \to I_B$ telle que $f_B(g(i)) = 1$ si et seulement si $f_A(i) = 1$.

Une réduction montre que si le problème B est décidable alors il en est de même du problème A. On utilise ensuite l'idée suivante : pour montrer qu'un problème B est indécidable, on choisit un problème A bien connu pour être indécidable, et l'on réduit A à B. On aura alors

$$B \text{ décidable} \Rightarrow A \text{ décidable} \qquad \text{et} \qquad A \text{ indécidable}$$

et l'on peut en déduire (règle de résolution !) que par conséquent B est indécidable. On notera que le raisonnement est tout aussi valide en remplaçant *décidable* par *récursivement énumérable.*

[8]

Corollaire 3.31. *Le langage* $L_u = \{\langle M\rangle\#w \mid M \text{ accepte le mot } w\}$ *n'est pas récursif. Plus précisément, son complément* $L_{\bar{u}}$ *n'est pas re.*

Démonstration. Nous alons réduire L_d à $L_{\bar{u}}$ en décrivant une procédure algorithmique pour transformer les instances de L_d en des instances de $L_{\bar{u}}$. Soit w une instanc de L_d.

1. la machine vérifie $w \in L_{enc}$ (lemme 3.27), si w n'est pas un encodage valide alors on retourne l'instance $\langle M_{palindrome}\rangle\#abba$ (on a bien $w \notin L_d$ et $\langle M_{palindrome}\rangle\#abba \notin L_{\bar{u}}$) ;
2. si $w = \langle M\rangle$ est un encodage valide, alors on retourne l'instance $w\#w = \langle M\rangle\#\langle M\rangle$ (on a bien $w \in L_d$ si et seulement si $\langle M\rangle\#\langle M\rangle \in L_{\bar{u}}$).

Cette réduction montre que si $L_{\bar{u}}$ est récursivement énumérable alors L_d l'est également, or le théorème 3.29 nous dit que L_d n'est pas récursivement énumérable, donc $L_{\bar{u}}$ non plus, et par conséquent (théorème 3.13) L_u n'est pas récursif. □

Corollaire 3.32. *Le langage* $L_{haltb} = \{\langle M\rangle \mid M \text{ s'arrête quand on la lance sur l'entrée vide}\}$ *n'est pas récursif.*

Démonstration. Nous alons réduire L_u à L_{haltb}. Etant donnée $\langle M\rangle\#w$ une instance de L_u, nous construisons l'instance $\langle M'\rangle$ suivante pour L_{haltb} :

7. Je vous souhaite d'être de cet avis un jour.
8. Navré pour le franglicisme, *many-one* qualifie la fonction g (de plusieurs vers un, il faut comprendre par là que ni l'injectivité ni la surjectivité ne sont imposées).

1. on vérifie $\langle M \rangle \in L_{enc}$ (lemme 3.27), si $\langle M \rangle$ n'est pas un encodage valide alors on retourne l'instance $\langle M' \rangle = \langle M \rangle$;
2. sinon on construit $\langle M' \rangle$ avec M' la machine qui commence par écrire w sur le ruban (cela est possible en utilisant $|w|$ états), puis entre dans l'état initial de M (M' va alors se comporter comme M), et nous rajoutons également à M' des transitions, depuis tous les états non finaux où M s'arrête (transition indéfinie), vers un état qui boucle à l'infini [9].

Dans tous les cas, nous avons bien $\langle M \rangle \# w \in L_u \iff \langle M' \rangle \in L_{haltb}$, donc si L_{haltb} est récursif alors L_u est récursif, or le théorème 3.31 nous dit que L_u n'est pas récursif, donc L_{haltb} n'est pas récursif. □

Voyez-vous la réduction utilisant le théorème 3.32 pour démontrer le théorème 3.28 ?

3.3.8 Théorème de Rice

Nous avons vu que de nombreuses questions sur les MT sont indécidables. Certaines questions sont clairement décidables, comme par exemple : est-ce qu'une MT donnée a 5 états ? Il s'avère cependant que **toute question non triviale qui concerne uniquement le langage reconnu par une MT (plutôt que la machine elle-même) est indécidable**. Une question non triviale étant une question qui n'est pas toujours vraie ou toujours fausse.

Définition 3.33. Soit P une famille de langages. On appelle P une **propriété non triviale** si il existe deux machines de Turing M_1 et M_2 telles que $L(M_1) \in P$ et $L(M_2) \notin P$.

Théorème 3.34. *Soit P une propriété non triviale. Il n'existe aucun algorithme pour décider si une MT M vérifie $L(M) \in P$.*

Démonstration. Nous utilisons une réduction depuis L_u. Sans perte de généralité, nous pouvons supposer $\varnothing \notin P$ (sinon on considère le complément de P au lieu de P). Puisque P est non triviale, il existe une MT M_P telle que $L(M_P) \in P$.

Etant donnée $\langle M \rangle \# w$ une instance de L_u, nous allons constuire l'instance $\langle M' \rangle$ (pour le problème d'appartenance de $L(M')$ à P) avec M' la machine qui :

1. copie son entrée u sur un ruban séparé pour l'utiliser plus tard (remarque 3.47) ;
2. écrit w sur le ruban et place la tête sur la première lettre de w ;
3. entre dans l'état initial de M. A partir de là M' simule M, en ignorant u, jusqu'à ce que M entre dans son état final ;
4. si M entre dans son état final, alors le mot u est recopié sur un ruban blanc (que des symboles B) et la machine M_P est simulée sur u. On entre dans un état final si M_P accepte u.

Etant donné n'importe quels $\langle M \rangle$ et w, la machine M' (et donc son code $\langle M' \rangle$) peut effectivement être construite algorithmiquement.

La correction de la réduction ($\langle M \rangle \# w \in L_u \iff L(M') \in P$) est assurée par les observations :
— si M accepte w, alors M' acceptera exactement les mots u que M_P accepte, donc dans ce cas $L(M') = L(M_P) \in P$;
— si M n'accepte pas w, alors M' n'accepte aucun mot u, donc dans ce cas $L(M') = \varnothing \notin P$.

On en conclut que si la propriété P est décidable alors L_u est décidable, or le théorème 3.42 nous dit que L_u n'est pas décidable, donc la propriété P n'est pas décidable. □

Remarque 3.35. M_1 **simule** M_2 = M_1 se comporte comme M_2 = M_1 suit la table de transition de M_2.

Appliquons le théorème de Rice. Les problèmes suivants sont indécidables :

9. Pour les fous de formalisme : soit $\langle M \rangle \# w$ une instance de L_u avec $M = (Q, \Sigma, \Gamma, \delta, q_0, B, q_F)$ et $w = w_0 \dots w_k$, dans le second cas nous construisons $\langle M' \rangle$ avec $M' = (Q \cup \{q'_0, \dots, q'_{k+1}\} \cup \{q'_{loop}\}, \Sigma, \Gamma, \delta', q'_0, q_F)$ avec $\delta'(q, a) = \delta(q, a)$ pour tout q et a où $\delta(q, a)$ est définie, et $\delta'(q'_i, B) = (q'_{i+1}, w_{k-i}, L)$ pour tout $0 \leq i \leq k$ afin d'écrire w sur le ruban initiallement vide, et $\delta'(q'_{k+1}, B) = (q_0, B, R)$ pour aller dans l'état initial de M sur le début du mot w, et $\delta'(q, a) = (q'_{loop}, B, R)$ pour tout $q \in Q$ et $a \in \Gamma$ pour lesquels $\delta(q, a)$ est indéfinie, et enfin $\delta'(q'_{loop}, a) = (q'_{loop}, B, R)$ pour tout $a \in \Gamma$.

— *est-ce qu'une MT donnée en entrée accepte tous les mots ?*
— *est-ce que $L(M)$ est un langage régulier pour une MT M ?*
— *est-ce qu'une MT donnée accepte tous les palindromes ?*

3.3.9 Réductions (version avec oracle)

Les réductions Turing que nous avons définies dans la section 3.3.7 ont une définition simple, mais sont en fait assez restraintes. Nous proposons maintenant une définition plus générale de la notion de réduction, qui correspond exactement à l'idée d'imaginer que nous puissions résoudre un problème B, et de s'intéresser alors à la résolution d'un problème A.

Définition 3.36. Un langage A est **Turing-réductible** à un langage B si il existe une machine de Turing avec **oracle** B, qui décide le langage A.

Une machine de Turing avec oracle B est une machine qui peut, au cours de son calcul, obtenir autant de réponses qu'elle souhaite sur des questions d'appartenance au langage B : est-ce que tel $w \in B$? est-ce que tel autre $w' \in B$? Et l'oracle pour le langage B lui donne des réponses oui/non, instantanément. La définition suivante explique comment implémenter formellement cette idée dans le modèle des machines de Turing.

Définition 3.37. Une machine de Turing M avec **oracle** B a un ruban supplémentaire appelé ruban d'oracle, et trois états spéciaux $q_{question}$, q_{oui} et q_{non}. A chaque fois que M entre dans l'état $q_{question}$, la machine va dans l'état q_{oui} (si $w \in B$) ou q_{non} (si $w \notin B$) avec w le contenu du ruban d'oracle. Les réponses aux questions d'appartenance à B sont données instantanément, et comptent comme une seule étape de calcul.

Remarque 3.38. Attention, dans la définition de MT avec oracle (et donc aussi dans la définition de Turing-réduction), on parle uniquement d'oracles récursifs, et plus de langages récursivement énumérables (car on obtient toujours une réponse oui/non de l'oracle).

Remarque 3.39. Une machine de Turing avec un oracle dont le langage est récursif, décide un langage qui est également récursif.

Remarque 3.40. Les réductions simples (many-one) et générales (avec oracle) ne sont pas équivalentes :
— si A est many-one-réductible à B, alors A est Turing-réductible (avec oracle) à B ;
— tout langage récursif est Turing-réductible à n'importe quel langage (puisque l'oracle est inutile), mais un langage récursif non vide ne peut pas être many-one-réduit au langage $B = \varnothing$ (puisque quels que soient g et i on aura toujours $f_B(g(i)) = 0$). Donc par exemple, le langage Σ^* est Turing-réductible au langage $\varnothing$, mais pas many-one-réductible ;
— tout langage est Turing-réductible à son complément, mais si A est many-one-réductible à B et B est récursivement énumérable (r.e.) alors A est également r.e. Par conséquent, le complément du problème de l'arrêt (qui n'est pas r.e.) n'est pas many-one-réductible à son complément (le problème de l'arrêt, qui est r.e.).

3.3.10 Indécidabilité de la logique du premier ordre

(source : http://kilby.stanford.edu/~rvg/154/handouts/fol.html)

Rappels sur la logique du premier ordre

Une logique du premier ordre est donnée par un langage (S_f, S_r). A Chaque symbole de fonction et prédicat est associé à une arité (les symboles de fonction d'arité 0 sont des constantes). Les termes sont définis récursivement comme suit :
— les variables sont des termes ;
— si $t_1, \ldots, t_n$ sont des termes et si f est une fonction n-aire, alors $f(t_1, \ldots, t_n)$ est un terme.
Les formules sont définies récursivement comme suit :
— si $t_1, \ldots, t_n$ sont des termes et si p est un prédicat n-aire, alors $p(t_1, \ldots, t_n)$ est une formule (atomique) ;

— si φ et ψ sont des formules, alors $\neg\varphi$, $\varphi \wedge \psi$, $\varphi \vee \psi$ et $\varphi \Rightarrow \psi$ sont des formules ;
— si φ est une formule et x une variable, alors $\forall x : \varphi$ et $\exists x : \varphi$ sont des formules.

Le théorème de complétude du calcul de la résolution pour la logique du premier ordre dit que si $\Gamma \models \varphi$ alors on peut dériver la close vide $\bot$ de $\Gamma \cup \{\neg\varphi\}$ (sous forme clausale, obtenue par mise sous forme prénexe puis skolémisation puis mise sous forme normale conjonctive des formules propositionnelles) avec les règles de factorisation et résolution.

Preuve d'indécidabilité

Théorème 3.41. *Le problème de savoir si $\Gamma \models \varphi$ dans la logique du premier ordre est indécidable.*

Démonstration. Nous allons réduire ce problème de décision à celui de décider le langage

$$L_u = \{\langle M\rangle \# w \mid M \text{ accepte l'entrée } w\}.$$

Etant donnés M et w, nous allons construire Γ et φ tels que $\Gamma \models \varphi$ si et seulement si $\langle M\rangle \# w \in L_u$. Puisque L_u est n'est pas récursif, on pourra en déduire qu'il ne peut pas exister d'algorithme pour décider si $\Gamma \models \varphi$ (sinon on pourrait construire un algorithme pour décider L_u, or le théorème 3.31 nous dit que c'est impossible).

Soit $M = (Q^M, \Sigma^M, \Gamma^M, \delta^M, q_0^M, B^M, q_F^M)$ et w une instance du problème d'appartenance à L_u. Nous définissons le langage $S = (S_f, S_r)$ avec $S_f = \{(\epsilon, 0)\} \cup \{(a, 1) \mid a \in \Gamma^M\}$ et $S_r = \{(f_q, 2) \mid q \in Q^M\}$: la constante ϵ, un symbole de fonction unaire a pour toute lettre $a \in \Gamma^M$, et un symbole de prédicat binaire f_q pour tout état q de la machine M. Voici l'idée que nous allons suivre : les variables x correspondront à des mots sur Γ^M, ϵ sera utilisé pour le mot vide, $a(w)$ sera le mot aw, et $f_q(x, y)$ signifiera que M, sur l'entrée w, peut atteindre la configuration $\bar{x}qy$ (avec $\bar{x}$ le mot miroir de x).

La formule

$$f_{q_0^M}(\epsilon, w)$$

correspond à la configuration initiale de la machine M sur l'entrée w, qui est atteignable, donc cette formule doit être vraie, nous la plaçons dans nos hypothèses Γ. Attention à la distinction entre le mot w (par exemple *abba*) et le terme de logique correspondant (dans cet exemple $a(b(b(a(\epsilon))))$, que nous écrirons aussi *abba* pour faciliter la lecture).

Nous allons maintenant faire en sorte que la formule

$$\varphi = \exists\, x\, \exists\, y : f_{q_F^M}(x, y)$$

soit vraie dans cette théorie (c'est-à-dire soit une conséquence logique de Γ) si et seulement si $\langle M\rangle \# w \in L_u$ (c'est-à-dire ssi M accepte w). Pour cela nous allons ajouter des formules dans Γ, qui nous permettrons de déduire que si une configuration est atteignable (le prédicat correspondant est conséquence logique de Γ), alors la configuration suivante donnée par la fonction de transition δ^M est également atteignable (le prédicat correspondant est conséquence logique de Γ).

Pour toute transition de la forme $\delta^M(q, a) = (p, b, R)$, nous rajoutons dans Γ la formule

$$\forall\, x\, \forall\, y : f_q(x, ay) \Rightarrow f_p(bx, y),$$

et pour toute transition de la forme $\delta^M(q, a) = (p, b, L)$, nous rajoutons dans Γ les formules

$$\forall\, x\, \forall\, y : f_q(cx, ay) \Rightarrow f_p(x, cby) \text{ pour tout } c \in \Gamma.$$

Pour être rigoureux nous devons aussi ajouter dans Γ les formules suivantes pour tout $q \in Q$:

$$\begin{array}{l} \forall\, x : f_q(x, \epsilon) \iff f_q(x, B) \\ \forall\, y : f_q(\epsilon, y) \iff f_q(B, y) \end{array}$$

La construction de Γ et φ est terminée. Puisque M possède un nombre fini de transitions sur un alphabet de taille finie, Γ est de taille finie.

Il nous reste à argumenter que $\Gamma \models \varphi$ si et seulement si M accepte le mot w. Pour le sens indirect, si l'on part de l'hypothèse que M accepte w alors il existe une suite de transition de la

machine de Turing M à partir de l'entrée w qui atteint l'état final q_F^M, et à chacune de ces transitions nous pouvons faire correspondre une application de la règle de résolution à partir de la formule $f_{q_0^M}(\epsilon, w)$, et puisque M sur l'entrée w atteint l'état final q_F^M alors nous allons pouvoir déduire que $\exists\, x\, \exists\, y : f_{q_F^M}(x, y)$, c'est-à-dire $\Gamma \models \varphi$. Pour le sens direct (qui est un peu plus compliqué à prouver formellement), si l'on part de l'hypothèse que $\Gamma \models \varphi$, alors il existe une suite d'application des règles de résolution et factorisation telle que l'on dérive φ de Γ, et de cette suite d'application nous pouvons en déduire une suite de transitions de la machine de Turing M à partir de l'entrée w (en fait, on peut se rendre compte que la règle de factorisation sera inutile, et que les applications de la règle de résolution partent de $f_{q_0^M}(\epsilon, w)$ pour arriver à φ en utilisant à chaque étape une formule qui correspond à une transition de la machine de Turing M). □

3.4 Universalité et complétude

Revenons sur le langage $L_u = \{\langle M\rangle \# w \mid M \text{ accepte l'entrée } w\}$. Nous avons vu que L_u n'est pas récursif. Cependant, L_u est re.

Théorème 3.42. *Le langage L_u est re.*

Démonstration. Plutôt que de décrire une machine de Turing qui reconnait L_u, nous nous contenterons de décrire informellement un semi-algorithme [10] pour déterminer si un mot est dans L_u.

Le semi-algorithme commence par vérifier si l'entrée a une forme correcte : le code $\langle M\rangle$ d'une machine, un symbole #, et un mot $w \in \{a, b\}^*$. Cette vérification peut effectivement être effectuée.

Ensuite, le semi-algorithme simule la machine M sur l'entrée w jusqu'à ce que (le cas échéant) la machine M s'arrête. Une telle simulation pas à pas peut effectivement être effectuée. Le semi-algorithme retourne alors la réponse « oui » si M s'arrête dans son état final. □

Nous pouvons en déduire les corollaires suivants.

Théorème 3.43. *Il existe des langages re qui ne sont pas récursifs, et la famille des langages re n'est pas close par complémentation.*

3.4.1 Universalité

Le théorème 3.42 nous dit qu'il existe une machine de Turing M_u capable de reconnaitre L_u (c'est-à-dire capable de dire « oui » pour un mot $w \in L_u$). Une telle machine M_u est appelée une **machine de Turing universelle** car elle peut simuler n'importe quelle machine sur n'importe quelle entrée, si on lui donne une description de la machine à simuler (le symbole # est un moyen de coder les couples d'entrées) :

$$M_u(\langle M\rangle, w) = M(w).$$

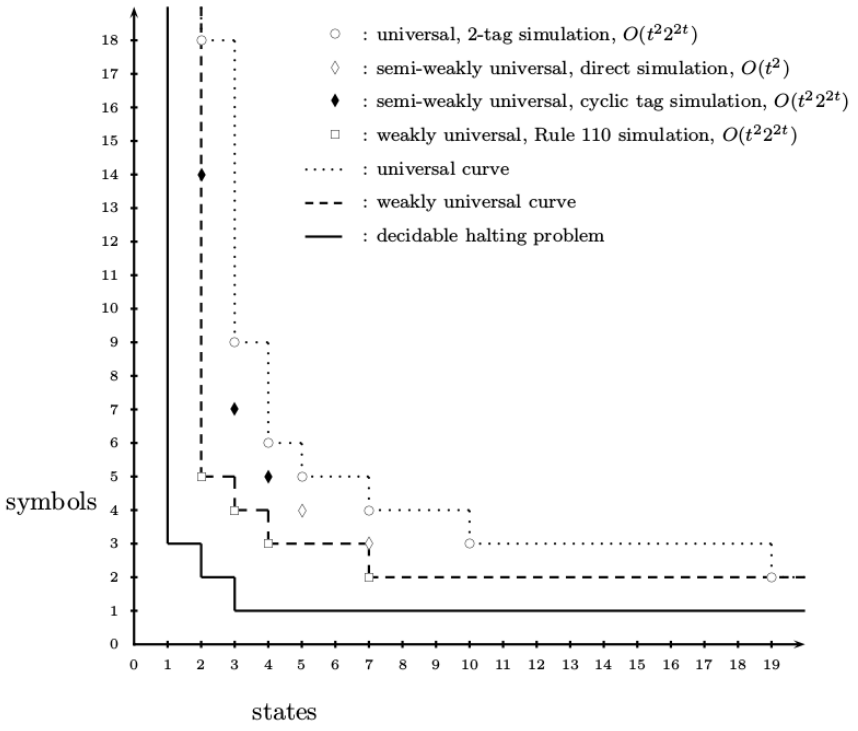
M_u est un ordinateur programmable : plutôt que de construire une nouvelle machine de Turing pour tout nouveau langage, on peut utiliser la même machine M_u et changer le **programme** $\langle M\rangle$ qui décrit quelle machine de Turing on souhaite simuler.

Ce concept est très important : imaginez si nous devions construire un nouvel ordinateur pour chaque algorithme que nous souhaiterions exécuter !

MT universelle	$\Longleftrightarrow$	ordinateur (système d'exploitation / interpréteur)
ruban	$\Longleftrightarrow$	disque dur
$\langle M\rangle$	$\Longleftrightarrow$	programme

Il est clair qu'une machine de Turing à 2 états et 2 symboles de ruban ne peut pas être universelle (pensez à son score au busy beaver). Trouver le plus petit nombre d'états et de symboles de ruban nécessaires à la construction d'une machine de Turing universelle est un problème compliqué, auquel ont travaillé Damien Woods et Turlough Neary [NW07]. La figure suivante est issue de la thèse de doctorat de ce dernier, soutenue en 2008.

10. C'est-à-dire un algorithme qui répond oui pour les mots appartenant au langage, mais qui peut ne pas répondre (ou répondre non) pour les mots n'appartenant pas au langage.



3.4.2 Turing-complétude

On appelle **modèle de calcul** la définition mathématique d'un ensemble d'opérations utilisables pour réaliser un calcul (une syntaxe et des règles décrivant la sémantique de la syntaxe). Exemple : les machines de Turing.

Définition 3.44. Un modèle de calcul capable de calculer toutes les fonctions calculables par des machines de Turing est appelé **Turing-complet**.

Remarque 3.45. Un modèle de calcul capable de simuler une machine de Turing universelle est Turing-complet.

Tous les langages de programmation que vous utilisez couramment (C, Haskell, Java, OCaml, Python...) sont bien entendu Turing-complets : si vous pouvez implémenter un simulateur de machines de Turing, alors le langages est capable de calculer toutes les fonctions calculables par des machine de Turing !

Petite liste de modèles, jeux et langages Turing-complets (parfois accidentellement) :
— le λ-calcul (très minimaliste, nous y reviendrons un peu plus loin),
— les fonctions μ-récursives,
— les pavages (puzzles), les automates cellulaires,
— les jeux Minecraft et Pokemon jaune (et de nombreux autres),
— Brainf*ck (un langage extrêmement simpliste qui comporte 8 instructions).
— Les briques de Lego^{TM}mécaniques (avec engrenages et pistons) : http://www.dailymotion.com/video/xrmfie/.

3.5 Thèse de Church Turing

Nous avons vu que les machines de Turing ne peuvent pas calculer toutes les fonctions, et même qu'elles ne sont capables d'en calculer qu'une infime partie. Il est légitime de se poser les questions suivantes : est-ce un bon modèle de calcul ? Ne pourrions nous pas définir un modèle de calcul qui puisse calculer un plus grand ensemble de fonctions ?

Définition 3.46. Deux modèles de calcul sont **équivalents** s'ils sont capables de se simuler mutuellement (donc ils calculent exactement le même ensemble de fonctions).

Remarque 3.47. Les MT non déterministes et multi-rubans sont équivalentes aux MT.

Il se trouve que tous les modèles de calcul « réalistes » qui ont été définis jusqu'à aujourd'hui sont équivalents aux machines de Turing : ils permettent de calculer exactement le même ensemble de fonctions. *Exactement* le même ensemble de fonctions !

Historiquement, en 1933 Kurt Gödel et Jacques Herbrand définissent le modèle des fonctions μ-récursives [11]. En 1936, Alonzo Church définit le λ-calcul (détails en section 3.5.3). En 1936 (sans avoir connaissance des travaux de Church), Alan Turing propose sa définition de machine. Church et Turing démontrent alors que ces trois modèles de calcul sont équivalents !

La thèse de Church-Turing, ou plutôt ses deux versions [Pé12], sont des énoncés que la communauté (dans sa majorité) pense vrais, mais qu'il n'est pas possible (du moins c'est le point de vue jusqu'à maintenant) de prouver. Ils énoncent que les machines de Turing capturent « correctement » la notion de calcul : toute autre façon de calculer, ou de définir le calcul, reviendrait au même [12].

3.5.1 Thèse de Church-Turing version physique

Thèse 3.48. *Toute fonction physiquement calculable est calculable par une MT.*

Autrement dit tout modèle de machine, qui peut effectivement exister selon les lois de la physique, sera au mieurx équivalent aux machines de Turing, sinon moins puissant (en terme d'ensemble de fonctions calculables).

Robin Gandy (qui a effectué son doctorat sous la direction d'Alan Turing) a travaillé sur cette question et démontré un résultat que nous reproduisons ci-dessous dans une formulation simplifiée [Gan80].

Théorème 3.49. *Toute fonction calculée par une machine respectant les lois physiques :*

1. *homogénéité de l'espace (partout les mêmes lois),*
2. *homogénéité du temps (toujours les mêmes lois),*
3. *densité d'information bornée (pas plus de n bits au m^2),*
4. *vitesse de propagation de l'information bornée (pas plus de c $m.s^{-1}$),*
5. *quiescence (configuration initiale finie et état de repos tout autour),*

est calculable par une machine de Turing.

Est-ce satisfaisant ? Une version quantique de ce théorème a été démontrée [AD11] (par un binôme dont l'un est désormais chercheur à Aix-Marseille Université).

3.5.2 Thèse de Church-Turing version algorithmique

Thèse 3.50. *Toute fonction calculée par un algorithme est calculable par une MT.*

Pas facile de définir ce qu'est, ou plutôt ce qu'en toute généralité pourrait être, un **algorithme**. On peut dire qu'un algorithme est exprimable par un programme rédigé dans un certain langage de

11. Les fonctions μ-récursives sont des fonctions de $\mathbb{N}^* \to \mathbb{N}$ définies à l'aide des briques de base suivantes : la fonction constante zéro, la fonction successeur, les projections qui renvoient leur $k^{\text{ième}}$ argument, un opérateur de composition des fonctions, un opérateur de récursion primitive ρ qui permet d'écrire des fonctions récursives, et un opérateur de minimisation μ qui permet de considérer le plus petit entier tel qu'une fonction retourne zéro ou tel qu'un prédicat donné est vrai.

12. Les machines quantiques n'échappent pas à la thèse de Church-Turing [AD11].

programmation, qu'il décrit des instructions pouvant être suivies sans faire appelle à une quelconque « réflexion ». Définir un modèle de calcul revient à définir une façon d'écrire des algorithmes.

Cette version de la thèse de Church-Turing est parfois appelée version symbolique, car elle exprime ce qu'il est possible de calculer à l'aide de symboles mathématiques auxquels on donne un sens calculatoire.

3.5.3 Le lambda calcul

Définition

En lambda calcul on définit des termes à base de variables x, de fonctions $\lambda x.x$, et d'application $x\ y$. Pour vous donner un aperçu, voici comment exprimer le nombre 2 : $\lambda s.\lambda z.s\ (s\ z)$; et l'addition a plus b : $\lambda a.\lambda b.\lambda s.\lambda z.a\ s\ (b\ s\ z)$.

La syntaxe des lambda termes est définie inductivement :

— x est un lambda terme, si x est une variable ;

— $\lambda x.t$ est un lambda terme, si t est un lambda terme et x une variable (lambda abstraction) ;

— $t\ s$ est un lambda terme, si t et s sont des lambda termes (lambda application).

Les lambda abstractions permettent de construire des fonctions, et les lambda application permettent de donner des arguments aux fonctions.

Les lambda termes peuvent être réduits en appliquant les règles suivantes :

— $(\lambda x.t)\ t' \mapsto t[t'/x]$ (beta réduction) ;

— $\lambda x.t \mapsto \lambda y.t[y/x]$ avec $y \notin FV(t)$ (alpha conversion) [13].

$t[t'/x]$ est le terme t dans lequel on a substituée toute occurrence de x par t'.

Les beta réductions permettent d'appliquer une fonction à un terme, et les alpha conversions permettent d'éviter les conflits entre les noms des variables.

Par exemple, quel que soit s on a $(\lambda x.x)\ s \mapsto x[s/x] = s$ montre que le terme $\lambda x.x$ est la fonction identité, et $(\lambda x.y)\ s \mapsto y[s/x] = y$ montre que le terme $\lambda x.y$ est une fonction constante. Le processus de réduction peut ne jamais terminer, par exemple

$$(\lambda x.x\ x)\ (\lambda x.x\ x) \mapsto (x\ x)[\lambda x.x\ x/x] = (x[\lambda x.x\ x/x])\ (x[\lambda x.x\ x/x]) = (\lambda x.x\ x)\ (\lambda x.x\ x).$$

On peut voir le lambda calcul comme une version idéalisée des langages de programmation fonctionnelle comme Haskell et OCaml, les beta réductions correspondant alors aux étapes de calcul. On peut avoir plusieurs façons d'appliquer des règles à un terme (de réduire un terme), le théorème de Church-Rosser nous dit que toutes les façons d'atteindre une forme normale beta (un terme non réductible) mènent au même terme.

Calculer en réduisant des lambda termes

(source : cours *Computability and lambda calculus* de Jacques Garrigues, 2013)

Le lambda calcul est très riche, on peut par exemple encoder les nombres, l'addition, la multiplication comme cela :

$$\begin{aligned}
\mathtt{n} &= \lambda f.\lambda x.(f\ \ldots(f\ x)\ldots) \\
\mathtt{+} &= \lambda m.\lambda n.\lambda f.\lambda x.(m\ f\ (n\ f\ x)) \\
\mathtt{*} &= \lambda m.\lambda n.\lambda f.(m\ (n\ f))
\end{aligned}$$

On peut encoder l'algèbre de Boole :

$$\begin{aligned}
\mathtt{t} &= \lambda x.\lambda y.x \\
\mathtt{f} &= \lambda x.\lambda y.y \\
\mathtt{not} &= \lambda b.\lambda x.\lambda y.(b\ y\ x)
\end{aligned}$$

et alors voici la fonction qui retourne `t` (true) ou `f` (false) suivant si son entrée vaut 0 ou non : $\mathtt{if0} = \lambda n.(n\ (\lambda x.\mathtt{f})\ \mathtt{t})$.

On peut créer des paires :

$$\begin{aligned}
\mathtt{pair} &= \lambda x.\lambda y.\lambda f.(f\ x\ y) \\
\mathtt{fst} &= \lambda p.(p\ \lambda x.\lambda y.x) \\
\mathtt{snd} &= \lambda p.(p\ \lambda x.\lambda y.y)
\end{aligned}$$

13. $FV(t)$ est l'ensemble des variables libres dans t (contrairement à celles liées par une lambda abstraction).

et alors $\texttt{fst}\ (\texttt{pair}\ a\ b) \mapsto \texttt{pair}\ a\ b\ \lambda x.\lambda y.x \mapsto (\lambda x.\lambda y.x\ a\ b) \mapsto a$.

En suivant l'idée des paires, on peut définir des n-uplets :

$$\lambda a_1.\ldots.\lambda a_n.\lambda f.(f\ a_1\ \ldots\ a_n)$$

et des listes :

$$\begin{array}{rcl} [] & = & \lambda x.\lambda y.y \\ [a_1, a_2, \ldots, a_n] & = & \lambda x.\lambda y.(x\ (\texttt{pair}\ a_1\ [a_2, \ldots, a_n])) \end{array}$$

Pour définir les listes infinies, nous avons besoin de l'opérateur de point fixe

$$\texttt{Y} = (\lambda f.\lambda x.(x\ (f\ f\ x)))\ (\lambda f.\lambda x.(x\ (f\ f\ x)))$$

qui est tel que $(\texttt{Y}\ t)$ se réduise en $(t\ (\texttt{Y}\ t))$. Alors une liste infinie de a sera représentée par le terme

$$[a, a, \ldots] = \texttt{Y}\ (\lambda x.(\texttt{pair}\ a\ x))$$

L'opérateur de point fixe `Y` permet aussi de définir des fonctions récursives dont le nombre d'itération est non borné, comme la fonction factorielle

$$\texttt{fact} = \texttt{Y}\ (\lambda f.\lambda n.\texttt{if0}\ n\ \texttt{1}\ (\texttt{x}\ n\ (f\ (\texttt{p}\ n))))$$

avec `p` la fonction predecesseur, dont la définition est laissée en exercice.

Lambda calcul et thèse de Church-Turing

Il est simple de se convaincre que les machines de Turing sont capables de simuler le lambda calcul : on peut écrire le code d'une machine de Turing (un programme) qui applique les règles de réduction à un terme écrit sur le ruban, jusqu'à ce que cela ne soit plus possible. On peut également traduire une machine de Turing M s'exécutant à partir d'une entrée w en un lambda terme, dont la réduction correspond à l'exécution de M sur le mot w. Une telle traduction applique (lambda application) les transitions (qui sont des lambda abstractions) à des configurations pour donner de nouvelles configurations (nous ne détaillerons pas une telle construction, qui demdande de bien comprendre l'opérateur de point fixe `Y`). Donc le modèle du lambda calcul est équivalent au modèle des machines de Turing.

3.6 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**... Nous pouvons faire 1 saut, 2 sauts, $\aleph_0$ sauts et même davantage ! Et il y aura toujours des fonctions non calculables.

C'est la morale de l'histoire, qui rappelle (et ce n'est pas un hasard [14]) le premier théorème d'incomplétude de Gödel :

Théorème 3.51. *Dans tout système formel cohérent et contenant l'arithmétique élémentaire, on peut construire un énoncé qui ne peut être ni prouvé ni réfuté dans cette théorie.*

14. En effet, il existe une correspondance (de Curry-Howard) entre preuve dans un système formel, et programme dans un modèle de calcul !

Bibliographie

[AD11] P. Arrighi and G. Dowek. The physical Church-Turing thesis and the principles of quantum theory. *arXiv :1102.1612*, 2011.

[Gan80] R Gandy. Church's Thesis and Principles for Mechanisms. *The Kleene Symposium*, 101 :123–148, 1980.

[Kar13] Jarkko Kari. *Automata and formal languages*. University of Turku, 2013. Course notes available at http://users.utu.fi/jkari/.

[NW07] Turlough Neary and Damien Woods. The complexity of small universal turing machines. pages 791–798, 2007.

[Pé12] M. Pégny. Les deux formes de la thèse de Church-Turing et l'épistémologie du calcul. *Philosophia Scientiae*, 16(3) :39–67, 2012.

[Sip06] Michael Sipser. *Introduction to the theory of computation*. Course Technology, 2006.