Calculabilité et Complexité (11X008) - Printemps 2025

4. Théorème de Rice, Diagonalisation, Non-reconnaissabilité

Enseignant: Arnaud Casteigts

Assistants: A. Berger, M. De Francesco
Moniteurs: E. Bussod, N. Beghdadi

Dans ce dernier cours sur la calculabilité, nous donnons quelques exemples supplémentaires de langages indécidables (et une réduction) puis nous présentons le *théorème de Rice*, qui établit qu'un grand nombre de propriétés sur les programmes sont indécidables. Nous présentons ensuite la technique de *diagonalisation* de Cantor, adapté à la construction d'un langage qui n'est pas reconnaissable. Enfin, nous évoquons rapidement la notion de *degrés de Turing*.

4.1 Autres langages indécidables

4.1.1 Langage universel

La semaine dernière, nous avons vu que le langage $L_U = \{(\langle M \rangle, w) \mid M \text{ accepte } w\}$, appelé le langage universel, est indécidable, car on peut réduire le langage de l'arrêt L_H à L_U. Notez cependant que L_U est *reconnaissable* : il suffit de simuler M sur l'entrée w et accepter si $M(w)$ accepte. En fait, c'est la machine universelle M_U qui reconnaît L_U, d'où son nom.

4.1.2 Langage vide

Étant donné une machine $\langle M \rangle$, on souhaite décider si cette machine n'accepte aucun mot, autrement dit, décider si son langage est vide : $L_\emptyset = \{\langle M \rangle \mid L(M) = \emptyset\}$.

Nous allons montrer que $L_\emptyset$ est indécidable. Pour cela, nous allons réduire L_U à $L_\emptyset$. Et comme L_U est indécidable, cela montre que $L_\emptyset$ l'est aussi.

Réduction : Supposons qu'il existe une machine $M_\emptyset$ qui décide $L_\emptyset$. Nous allons utiliser $M_\emptyset$ pour construire une machine M_U qui *décide* L_U (cette machine sera différente de la machine M_U habituelle, qui ne fait que reconnaître L_U).

Nous donnons ici la réduction sous forme d'un programme en pseudo-code. Ce programme définit une fonction interne et suppose qu'il peut accéder à la description de cette fonction. Cela peut sembler bizarre, mais c'est tout à fait valide (dans le sens où ce type de choses peuvent être réalisées par (et sur) des machines de Turing).

```
def M_U(⟨M⟩, w) :
    def M'(x) :
        if x == w :
            return M(x)
        else :
            reject
    if M_∅(⟨M'⟩) accepts :   //L(M) ≠ ∅
        reject
    else :                   //L(M) = ∅
        accept
```

En entrée, M_U prend une machine $\langle M \rangle$ et une entrée w et on veut décider si M accepte w. On commence par créer une machine intermédiaire M' qui rejette toutes les entrées sauf w, et si l'entrée est w, elle renvoit $M(w)$. Observons qu'on a alors $L(M') \neq \emptyset$ si et seulement si M accepte w. On peut donc utiliser maintenant $M_\emptyset$ pour savoir si $L(M')$ est vide ou non, et conclure selon la réponse que M accepte w ou non.

4.1.3 Le problème du langage non-vide

Étant donné la description de machine $\langle M \rangle$, on souhaite décider si M accepte au moins un mot, autrement dit, on s'intéresse au langage complément de $L_\emptyset$:

$\overline{L_\emptyset} = \{\langle M \rangle \mid L(M) \neq \emptyset\}$.

Le fait que $L_\emptyset$ soit indécidable implique que $\overline{L_\emptyset}$ l'est aussi. (Par l'absurde, si $\overline{L_\emptyset}$ était décidable, alors $L_\emptyset$ serait décidable... souvenez-vous que la famille des langages décidables est stable par complémentation.) Vous verrez en exercice que $\overline{L_\emptyset}$ est cependant reconnaissable, et on peut déduire de cela que $L_\emptyset$ ne peut pas être reconnaissable. (Pourquoi ? Réfléchissez-y par l'absurde à nouveau.)

4.1.4 Problème de correspondance de Post

Le problème de correspondance de Post (PCP) est un problème plus naturel que les précédents, car il ne s'agit pas d'un langage de description de machines. Étant données deux listes de k mots chacune,

$$A = x_1, \ldots, x_k \qquad B = y_1, \ldots, y_k,$$

est-t-il possible de construire un mot w en concaténant certains mots de A, de sorte que la concaténation des mots de B *ayant les mêmes indices* donnent aussi w. Les répétitions sont autorisées.

Par exemple, prenons $A = (\texttt{b}, \texttt{a}, \texttt{ca}, \texttt{abc})$ et $B = (\texttt{ca}, \texttt{ab}, \texttt{a}, \texttt{c})$. On peut trouver une solution qui correspond à la suite d'indices $2, 1, 3, 2, 4$. Cela donne :

`a.b.ca.a.abc` (pour A)

`ab.ca.a.ab.c` (pour B)

En revanche, pour $A = (ba, abb, bab)$ et $B = (bab, bb, abb)$, il n'existe aucune solution.

Le problème de correspondance de Post est celui de décider s'il existe une solution, autrement dit à décider le langage :

$L_P = \{A = x_1, \ldots, x_k; B = y_1, \ldots, y_k \mid \exists i_1, i_2, ..., i_\ell,\ x_{i_1}x_{i_2}...x_{i_\ell} = y_{i_1}y_{i_2}...y_{i_\ell}\}$.

Ce problème est indécidable. Nous ne ferons pas la preuve, mais c'est bon à retenir, car de nombreux problèmes ont été montrés indécidable en utilisant une réduction depuis PCP.

4.2 Théorème de Rice

Le théorème de Rice a des implications très profondes en informatique. Intuitivement, ce théorème établit qu'on ne peut pas vérifier, en général, qu'un programme informatique est correct ou qu'il effectue bien le traitement que l'on souhaite qu'il fasse.

Theorème 4.1 (Rice)**.** *Toute propriété sémantique non triviale d'un programme est indécidable*

Décryptons. Déjà, on peut remplacer "programme" par "machine de Turing". Puis :

- **Propriété sémantique** : Il s'agit d'une propriété P qui ne concerne pas les détails de la machine elle-même (la syntaxe), mais seulement le langage qu'elle reconnaît (la sémantique). Concrètement, si M_1 et M_2 sont deux machines telles que $L(M_1) = L(M_2)$, alors soit elles satisfont toutes deux P, soit aucune des deux ne satisfait P.
- **Propriété non triviale** : Il faut qu'il existe au moins une machine M qui satisfait P et une machine M' qui ne satisfait pas P.

Si ces deux conditions sont vérifiées, alors le théorème de Rice nous dit que le langage $L = \{\langle M\rangle \mid M \text{ satisfait } P\}$ est indécidable.

Remarque 4.2. Il existe un autre théorème de Rice qui permet de montrer qu'un langage n'est pas reconnaissable. Ce théorème est plus compliqué et n'est pas au programme.

4.2.1 Exemples

Le langage vide (revisité)

$L_\emptyset = \{\langle M\rangle \mid L(M) = \emptyset\}$ (M n'accepte aucun mots)

Nous avons déjà montré l'indécidabilité de $L_\emptyset$. Le théorème de Rice permet d'obtenir le même résultat plus facilement. Ici, la propriété P est $L(M) = \emptyset$.

- P est-elle une propriété sémantique ?
  $\rightarrow$ Soient M_1 et M_2 telles que $L(M_1) = L(M_2)$. Clairement, $L(M_1) = \emptyset$ si et seulement si $L(M_2) = \emptyset$ (puisque $L(M_1) = L(M_2)$). P est donc bien une propriété de langage.

- P est-elle une propriété non triviale ?
  $\rightarrow$ On peut facilement créer une machine M_1 telle que $L(M_1) = \emptyset$ et une autre machine telle que $L(M_2) \neq \emptyset$. P est donc une propriété non triviale.

Le théorème de Rice nous permet de conclure que $L_\emptyset$ est indécidable.

Le langage infini

Peut-on décider si une machine accepte un nombre infini de mots ?

$L_\infty = \{\langle M \rangle \mid |L(M)| = \infty\}$

La propriété P est ici $|L(M)| = \infty$.

- Propriété sémantique ? Soit M_1 et M_2 deux machines telles que $L(M_1) = L(M_2)$. Clairement, $|L(M_1)| = \infty$ si et seulement si $|L(M_2)| = \infty$. Donc oui, P est sémantique.
- Propriété non triviale ? Oui (même exemple que pour $L_\emptyset$).

L_∞ est donc indécidable.

Vous vous demandez certainement à quoi pourrait ressembler une propriété non-sémantique ? Par exemple, pour le langage $L = \{\langle M \rangle \mid$ la machine M a 5 états$\}$, la propriété qui nous intéresse n'est pas sémantique : elle ne porte pas sur le langage $L(M)$. Et d'ailleurs, ce langage est tout à fait décidable.

4.2.2 Portée du théorème et intuition

Prenons un peu de recul. Imaginons que vous travaillez chez Airbus et vous avez demandé à l'équipe de développement de créer un module qui effectue un certain traitement. Par exemple, une fonction qui calcule le carré d'un nombre (oui, c'est caricaturalement simple). Vous souhaitez vérifier que le programme développé calcule bien le carré, dans tous les cas et sans erreur. Et bien le théorème de Rice nous dit que c'est impossible à vérifier : il existe certains programmes pour lesquels vous ne pouvez pas décider cela. Par exemple, vous pouvez imaginer un programme qui effectue un traitement compliqué (dont on ne sait pas s'il termine, par exemple), puis renvoie le carré de son entrée. Décider si ce programme renvoit le carré revient à décider la partie compliquée du programme. Malheureusement, ce type de "contamination" se produit réellement en pratique, dès qu'un programme devient suffisamment complexe. C'est pour cela que nos compilateurs ne peuvent généralement pas détecter que l'exécution d'un programme posera problème.

4.3 Diagonalisation et non-reconnaissabilité

Lorsque l'on parle d'infini, beaucoup de choses deviennent étranges. Par exemple, y a-t-il plus d'entiers relatifs ($\mathbb{Z}$) que d'entiers naturels ($\mathbb{N}$) ? Intuitivement, il y en a le double, mais $2 \times \infty = \infty$, comment s'y retrouver ? Pour montrer que deux ensembles infinis ont la même taille (on parle de *cardinalité*), il suffit établir une *bijection* entre les deux, c'est à dire mettre en correspondance leurs éléments deux à deux. Par exemple, pour les entiers naturels et les entiers relatifs, on peut les associer comme suit :

0	1	2	3	4	5	6	$\cdots$
0	1	-1	2	-2	3	-3	$\cdots$

On voit bien ici que pour chaque entier naturel i, il existe un $i^{\text{ème}}$ entier relatif, et réciproquement, chaque entier relatif correspond à exactement un entier naturel i. Les deux ensembles ont donc bien la même cardinalité, qui est celle des entiers naturels, aussi appelée $\aleph_0$ ("aleph 0") ou **infini dénombrable**. On dit aussi qu'un tel ensemble peut être *énuméré*. Qu'en est-il des nombres rationels $\mathbb{Q}$? Ils peuvent aussi être énumérés, ils ont donc la même cardinalité que les entiers. Qu'en est-il des réels $\mathbb{R}$?

En 1874, le mathématicien Georg Cantor démontra que l'ensemble $\mathbb{R}$ n'est pas dénombrable, il est strictement plus grand que $\mathbb{N}$. Il s'agit d'une preuve par l'absurde utilisant un argument de *diagonalisation*. Supposons que $\mathbb{R}$ est dénombrable. Il existe donc un ordre dans lequel on peut énumérer ses éléments. En s'appuyant sur un tel ordre, on peut ensuite montrer qu'il existe nécessairement des nombres qui ne seront jamais atteints, ce qui est une contradiction. L'idée est la suivante, supposons qu'il existe un ordre pour énumérer les réels, par exemple :

```
Natural    Real
    0    0.236436775676...
    1    0.098473294543...
    2    0.193214042202...
    3    0.843279242093...
    4    0.012934812343...
    5    0.639423412934...
    6    0.017773923845...
    7    0.238920090909...
    8    0.123984732999...
    9    0.646329878122...
   10    0.000123943437...
   11    0.981298312892...
    .          .
    .          .
    .          .
         0.293233992132...
         0.746894310875...
```

Intéressons-nous au nombre formé par la diagonale comprenant la $i^{\text{ème}}$ décimale de la $i^{\text{ème}}$ ligne (en rouge sur le dessin), par exemple ici $x = 0.293233992132\ldots$. On peut alors

construire un nombre y dont les décimales sont toutes différentes des décimales de x au même endroit (première décimale différente de 2, deuxième différente de 9, etc.), par exemple $y = 0.746894310875\ldots$. Et bien un tel nombre ne sera jamais atteint par l'énumération, car si c'était le cas, par exemple à la ligne j, sa $j^{\text{ème}}$ décimale serait la même que celle de la diagonale... ce qui est exclu par construction. En fait, il y a plein de choix possibles pour y, et donc plein de nombres qui ne seront jamais atteints.

Cet argument peut être appliqué quel que soit l'ordre choisi. L'ensemble $\mathbb{R}$ n'est donc pas dénombrable. On dit que sa cardinalité est $2^{\aleph_0}$ (**infini non-dénombrable**).

En quoi cela nous concerne ? La notion d'énumération et de diagonalisation sont très utiles pour étudier les limites de la calculabilité. Voyons cela.

4.4 Énumération des machines de Turing

Lors du dernier cours, nous avons vu que toute machine de Turing peut être décrite sous forme textuelle sur l'alphabet $\{0, 1\}$ (son encodage). Outre la simulation, cet encodage permet également d'énumérer les machines de Turing.

Comment faire ? Tout d'abord, remarquons qu'on peut énumérer tous les mots binaires, par exemple dans l'ordre "shortlex" qui liste les mots par ordre de taille ("short") et par ordre lexicographique à l'intérieur d'une même taille ("lex"), autrement dit : $\mathtt{0}, \mathtt{1}, \mathtt{00}, \mathtt{01}, \mathtt{10}, \mathtt{11}, \mathtt{000}, \ldots$. On peut coupler cette énumération à une procédure qui vérifie si un mot donné représente une machine valide, à savoir (c.f. cours précédent) commence-t-il et termine-t-il par 111, chaque transition est-elle bien au format $u\mathtt{1}v\mathtt{1}x\mathtt{1}y\mathtt{1}z$, *etc.* On peut ainsi énumérer toutes les machines de Turing possibles.

4.5 Un langage non-reconnaissable (langage diagonal)

Nous allons utiliser un argument diagonal pour montrer que certains langages ne sont pas reconnaissables. Soit $M_1, M_2, \ldots$ une énumération des machines de Turing. On note L_i le langage *reconnu* par M_i, autrement dit, l'ensemble des mots w que M_i accepte.

Faisons un tableau dont les colonnes correspondent aux machines de Turing M_i et les lignes correspondent à tous les mots w_j (par exemple, dans l'ordre shortlex). La case (i, j) contient 1 si M_i accepte w_j et 0 si elle rejette ou ne termine pas

Mots \ Machine	M_1	M_2	M_3	M_4	M_5	...
ε	**0**	1	0	1	...	
0	0	**1**	1	0	...	
1	0	0	**0**	0	...	
00	1	0	1	**0**	...	
01	0	1	0	1	...	
10	1	0	0	0	...	
11	1	0	0	0	...	
000	0	1	0	1	...	
...	...					

Inspiré par l'argument de Cantor, on peut définir un langage diagonal L_{diag} qui diffère en chaque point de la diagonale. Autrement dit,

$L_{diag} = \{w_i \mid M_i \text{ n'accepte pas } w_i\}$. Ici, par exemple, $L_{diag} = \{\varepsilon, 1, 00, \cdots\}$.

Par l'absurde, soit M_k une machine qui reconnaît ce langage. Observons le comportement de M_k sur le mot w_k. Si la valeur de la cellule (k, k) valait 1, alors par construction ce mot n'est pas dans L_{diag}, mais la machine M_k l'accepte. Si elle valait 0, alors ce mot est dans L_{diag}, mais la machine ne l'accepte pas. M_k doit donc se tromper au moins sur le mot w_k. L_{diag} n'est donc pas reconnaissable.

Remarque : Certes, dans l'absolu, on aurait définir un *autre* ordre d'énumération des machines telle que ce langage là est reconnaissable. Mais dans ce cas, un autre langage L_{diag} correspondant à la définition ci-dessus n'aurait pas été reconnaissable. La subtilité est que l'on définit L_{diag} une fois l'ordre d'énumération fixé. Et pour tout ordre d'énumération, un tel langage existe.

En résumé, quel que soit l'ordre considéré pour énumérer des machines de Turing et l'ordre utilisé pour énumérer les mots, il existe des langages qu'aucune machine ne reconnaît. Plus généralement, on peut montrer que l'ensemble des langages n'est pas dénombrable, il a la cardinalité des réels $2^{\aleph_0}$, tandis que l'ensemble des machines est dénombrable (cardinalité $\aleph_0$). Il y a donc beaucoup plus de langages que de machines [1].

4.6 Degrés de Turing

Soient deux langages L_1 et L_2. Ces langages sont **Turing-équivalents** s'il existe une réduction de L_1 à L_2 et une réduction de L_2 à L_1. Autrement dit, si $L_1 \leq_T L_2$ et $L_2 \leq_T L_1$. On note cela $L_1 \equiv_T L_2$.

Tous les problèmes décidables sont Turing-équivalents. Cette déclaration est vraie, bien

1. "Beaucoup plus de problèmes algorithmiques que d'algorithmes".

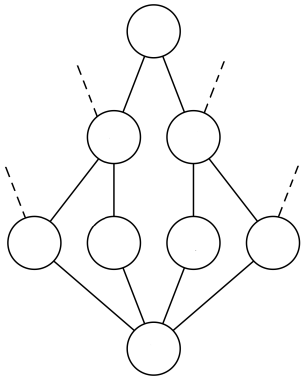
qu'elle ne donne aucune information intéressante. En effet, si L_1 et L_2 sont décidables, alors L_1 peut être décidé en ayant accès à une machine pour L_2, mais c'est tout aussi vrai sans L_2 (puisque L_1 est déjà décidable). Même remarque dans l'autre sens.

Plus intéressant : Les problèmes indécidables sont-ils tous équivalents ?

Non ! Certains langages indécidables sont utiles pour certains autres, mais pas pour tous. Étant donné deux langages indécidables L_1 et L_2, on peut être dans l'un des quatre cas suivants :

- $L_1 \equiv_T L_2$ (ils sont Turing-équivalents)
- $L_1 \leq_T L_2$ mais l'inverse n'est pas vrai (L_2 est strictement plus fort)
- Idem dans l'autre sens (L_1 est strictement plus fort)
- L_1 et L_2 sont incomparables (aucun des deux ne se réduit à l'autre)

Ces possibilités permettent de classifier les langages par ordre de difficulté, chaque ordre (appelé un **degré de Turing**) regroupant des langages Turing-équivalents. Du fait de l'incomparabilité entre certains langages, il s'agit seulement d'un ordre partiel, tel qu'illustré ci-dessous, où le degré le plus bas correspond aux langages décidables (le nombre de degrés représentés et leurs relations sont complètement arbitraires).



La classification de ces degrés a été un domaine de recherche très actif à partir des années 1950 (il l'est moins aujourd'hui).