Calculabilité, décidabilité.

Dernière mise à jour le : 28/05/2024

... ou l'étude des possibilités et limites des algorithmes

Introduction

Le problème de décision de Hilbert

La notion de **calculabilité**, que l'on définira plus bas, mérite de se pencher tout d'abord un petit peu sur son origine historique : le **problème de décision de Hilbert**.

Le **problème de décision**, ou *Entscheidungsproblem* en allemand, est une question posée en 1928 par le mathématicien allemand David Hilbert :

> Existe-t-il un algorithme capable de déterminer, pour tout énoncé mathématique bien formulé donné en entrée, si ce dernier est vrai ou faux ?
>
> ***Le problème de décision de Hilbert***

David Hilbert (1862-1943)

Crédits : See page for author, Public domain, via Wikimedia Commons

C'est cette question qui a conduit à définir de manière plus précise ce qu'est un algorithme et ce que peut, ou ne peut pas, calculer un algorithme. Autrement dit, c'est le problème de décision de Hilbert qui est à l'origine de la théorie de la **calculabilité**.

En 1936, Alonzo Church et Alan Turing démontrent de manière totalement indépendante que la réponse au problème de décision est négative.

La vidéo ci-dessous présente ce problème ainsi que les théories informatiques de Church et Turing (et Gödel) qui en ont découlées.

Source vidéo : https://youtu.be/Zci9m08HQws

Programme en tant que donnée

L'une des idées clés dans la théorie de la calculabilité, que nous aborderons par la suite, est de considérer un programme informatique comme une donnée manipulable par d'autres programmes. Concrètement, cela signifie qu'un programme peut être traité comme une suite de bits que d'autres programmes peuvent lire, écrire et exécuter.

Exemples

Pour exécuter un fichier python `prog.py` on peut utiliser la commande `python` (ou `python3`) dans un terminal de la manière suivante :

```
python prog.py
```

Mais derrière la commande `python` il y a un programme, et ce dernier prend donc ici un autre programme (`prog.py`) en entrée et le lance, procède à des verifications syntaxiques (indentation, nommage, etc.), puis exécute et affiche ce qui est demandé dans `prog.py` et s'arrête (si tout va bien). Cette commande `python` est appelée **interpréteur Python**, qui est donc un programme qui s'applique à des programmes Python.

De même, les langages compilés utilisent un **compilateur** qui n'est autre qu'un programme qui prend en paramètre un programme pour le convertir en langage machine.

Voici quelques autres exemples simples :

- Pour télécharger un logiciel on utilise un navigateur ou un gestionnaire de téléchargement qui sont eux-mêmes des programmes.
- Un système d'exploitation est un programme très complexe qui gère et fait fonctionner tous les autres programmes.
- Un logiciel antivirus utilise un programme antivirus pour scanner d'autres programmes afin d'y trouver des séquences suspectes.
- Un virus lui-même, est un programme qui prend en entrée des programmes et qui en produit d'autres, infectés.
- etc.

Et même en Python...

Dans le même ordre d'idée, on déjà écrit des fonctions qui prenaient d'autres fonctions en paramètres. Par exemple, la fonction `sorted` de Python peut prendre en paramètre - nommé `key` - une fonction à appliquer à chaque élément et dont le résultat permet les comparaisons et, au final, le tri. En passant la fonction `len` en paramètre à la fonction `sorted` on obtient un tri par longueur des éléments :

```
>>> animaux = ['veau', 'vache', 'cochon', 'anaconda', 'chat', 'chien', 'ver',
               'poule', 'souris']
>>> sorted(animaux, key=len)  # ici la fonction len est passée en paramètre
['ver', 'veau', 'chat', 'vache', 'chien', 'poule', 'cochon', 'souris', 'anaconda']
```

De même, on a déjà utilisé le module `doctest` (en classe de Première, chapitre *Documenter et tester un programme*) qui, grâce à la fonction `testmod()`, permet de lancer les tests des fonctions du module courant : c'est donc une fonction qui prend en entrée d'autres fonctions et les exécute pour vérifier leur comportement.

Calculabilité

La notion d'*algorithme* est très ancienne, elle désigne au sens large des méthodes de calculs. Un algorithme attend des entrées et calcule un résultat en fonction des entrées. Ainsi, un algorithme peut être vu comme une fonction mathématique.

La **théorie de la calculabilité** s'intéresse aux fonctions qui peuvent être calculées par un algorithme. Savoir ce qui est calculable et ce qui ne l'est pas permet de voir les limites des problèmes que peuvent résoudre les ordinateurs.

> Une **fonction** est dite **calculable** si elle peut être calculée par un algorithme. Autrement dit, si nous pouvons écrire un programme qui, pour toute entrée donnée, produira la sortie correcte en un nombre fini d'étapes.

Par exemple, la fonction qui prend en entrée deux nombres et produit leur somme est calculable, car nous pouvons écrire un algorithme (un programme) qui calcule la somme de deux nombres.

Cette notion intuitive a été formalisée par Alonzo Church et Alan Turing dans les années 1930 pour déterminer de manière plus précise ce qui peut, ou ne peut pas, être calculé par un algorithme.

Voyons de manière succincte (ce n'est pas du tout au programme) leurs deux approches considérablement différentes, mais qui se sont révélées être équivalentes, et surtout les implications de leurs travaux sur l'informatique théorique.

La vision de Church

Alonzo Church (1903-1995)
Crédits : Princeton University, Fair use, Link

L'américain Alonzo Church a proposé de définir les fonctions calculables comme des fonctions dont le résultat est obtenu par une combinaison d'applications de fonctions calculables. Il a défini un langage, appelé le **λ-calcul** (à prononcer « lambda-calcul »), dans lequel on manipule des expressions appelées **λ-expressions** qui permettent de décrire les fonctions calculables qui sont alors des fonctions prenant en paramètres d'autres fonctions calculables et dont le résultat est lui-même une fonction calculable.

Ainsi, pour Church, une fonction est calculable si elle peut s'exprimer comme une λ-expression.

Dans la théorie de Church, tout est fonction. C'est un peu comme si on écrivait un programme Python uniquement en manipulant la construction `lambda` (voir le chapitre sur la programmation fonctionnelle). Cette théorie est à la base de ce qui constituera les langages de *programmation fonctionnelle* apparus des dizaines d'années plus tard (avec LISP, puis ML, Haskell, etc.).

Exemples

Les exemples proposés sont là uniquement pour vous donner une idée de ce qu'est le λ-calcul. Ils ne sont ni à connaître, ni même à comprendre !

Le λ-calcul manipule ce qu'on appelle des expressions. Par exemple :

- les entiers 0 et 1 correspondent respectivement aux expressions $\lambda f.\, \lambda x.\, x$ et $\lambda f.\, \lambda x.\, f(x)$.
- la fonction $f : x \longmapsto x + 1$ est représentée par l'expression : $\lambda x.\, \lambda f.\, \lambda z.\, x(f)(f(z))$.

On peut écrire cela en Python en utilisant uniquement la construction `lambda` :

```
zero = lambda f: lambda x: x
un = lambda f: lambda x: f(x)
ajouter_un = lambda x: lambda f: lambda z: x(f)(f(z))
```

Le nombre 2 peut alors être obtenu de la manière suivante :

```
deux = ajouter_un(un)
```

Dans ce cas, `deux` est une fonction et non un nombre entier comme nous les connaissons en programmation classique. Pour vérifier que `deux` correspond bien à notre nombre entier 2, il va falloir "extraire" une valeur numérique de la fonction `deux`. Plus précisément, `deux` est ici une fonction qui applique une autre fonction `f` à un argument `x`. Elle s'utilise en écrivant `deux(f)(x)` et devrait normalement appliquer *deux* fois (c'est ce qu'on veut vérifier !) la fonction `f` à l'argument `x` (on veut vérifier que `deux(f)(x)` correspond à $f(f(x))$). Pour vérifier, on peut prendre comme fonction `f` la fonction `triple` suivante :

```
>>> triple = lambda x: x * 3
```

Et on voit bien que la fonction `triple` est bien appliquée deux fois dans chaque cas :

```
>>> deux(triple)(1)  # calcule bien 1 * 3 * 3
9
>>> deux(triple)(2)  # calcule bien 2 * 3 * 3
```

```
18
>>> deux(triple)(5)  # calcule bien 5 * 3 * 3
45
```

On retrouve dans la théorie du λ-calcul le fait qu'une fonction est une donnée comme une autre, qui peut être passée en paramètre à une fonction et aussi le résultat d'une fonction.

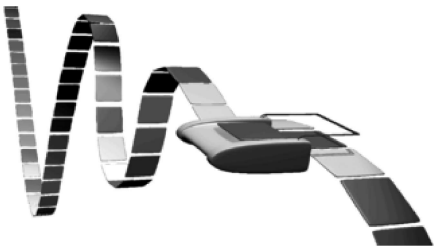
Le λ-calcul peut être considéré comme un ancêtre des langages de programmation, bien avant la création des ordinateurs.

La vision de Turing

Alan Turing (1912-1954)

Crédits : Auteur anonyme, Public domain, via Wikimedia Commons

Le britannique Alan Turing a développé quant à lui, à la même époque, une approche plus mécanique du calcul en définissant ce qu'on appelle une **machine de Turing**.



Vue d'artiste d'une machine de Turing

Crédits : Schadel (http://turing.izt.uam.mx), Public domain, via Wikimedia Commons

Bien que le terme *machine* puisse laisser penser le contraire, une machine de Turing est un concept *abstrait* (de telles machines ont été fabriquées et il existe de nombreux simulateurs, mais cela n'a pas vraiment d'intérêt concret). En simplifiant un peu, une telle machine est constituée :

- d'un **ruban** infini de cases consécutives qui peuvent chacune contenir un symbole d'un alphabet donné (par exemple : 0, 1 et vide)
- d'un **ensemble de règles** simples qui indiquent à la machine, qui se déplace le long du ruban, quel symbole écrire sur le ruban, comment déplacer la tête de lecture (vers la gauche ou la droite) et quel est le nouvel état, en fonction du symbole lu sur le ruban et de l'état courant de la machine.

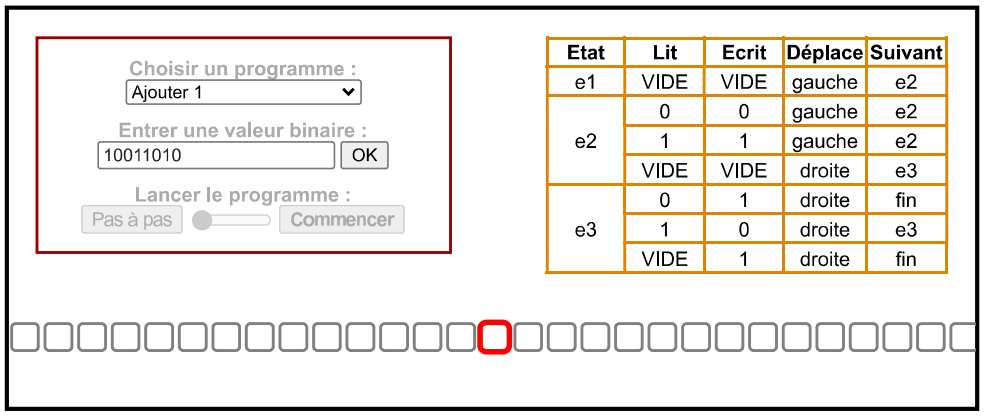
Cet ensemble de règles peut être vu comme le programme de cette machine, donc comme un algorithme ou comme l'expression d'une certaine fonction f : si on écrit au départ sur le ruban un paramètre e et qu'on démarre la machine, la valeur lue sur le ruban lorsqu'elle s'arrête est le résultat $f(e)$.

Selon Turing, une fonction est calculable s'il existe une machine de Turing pouvant calculer ses valeurs.

Exemples

Les exemples proposés sont intéressants pour votre culture et permettent de visualiser le fonctionnement d'une machine de Turing. Ils ne sont ni à connaître, ni même à comprendre !

L'animation suivante propose de visualiser le fonctionnement d'une telle machine pour certains programmes. Il peut être intéressant de comprendre le premier programme, l'algorithme qui ajoute 1 à un nombre binaire (qui prouve donc que la fonction $f : x \longmapsto x + 1$ est calculable) :



Etat	Lit	Ecrit	Déplace	Suivant
e1	VIDE	VIDE	gauche	e2
e2	0	0	gauche	e2
	1	1	gauche	e2
	VIDE	VIDE	droite	e3
e3	0	1	droite	fin
	1	0	droite	e3
	VIDE	1	droite	fin

Crédits : Animation HTML5/JS réalisée par Hugo Lehmann, Centrale Lille Projets, librement adaptée d'une applet Java écrite par Hamdi Ben Abdallah.

On peut penser que chaque machine est monotâche puisque l'ensemble de règles diffère d'une machine à l'autre. Mais Turing démontre ensuite qu'il existe des **machines de Turing universelles** qui prennent en entrée une autre machine de Turing M et une entrée e pour la machine M et qui simulent l'exécution de la machine M sur l'entrée e.

On retrouve également ici, le principe qu'un programme, ici représenté par une machine de Turing, est une donnée comme une autre, puisqu'il peut être donné en entrée à un autre programme (à une autre machine de Turing).

Étant capable de simuler n'importe quelle autre machine de Turing, une machine de Turing universelle peut être considérée comme l'ancêtre des ordinateurs programmables, qui eux aussi peuvent effectuer les calculs de n'importe quelle autre machine.

Universalité de la calculabilité

La **thèse de Church** (aujourd'hui rebaptisée **thèse de Church-Turing**) énoncée par Church en 1936, stipule que les fonctions pouvant être définies comme des fonctions λ-calculables de Church et les fonctions pouvant être calculées par des machines de Turing sont exactement les mêmes. Autrement dit, Church et Turing, bien qu'ayant proposé des définitions totalement différentes, ont en réalité définit la même notion de fonction calculable, et donc d'algorithme.

Cette notion commune sert aujourd'hui d'étalon. En particulier, tous les langages de programmation, quels que soient leur paradigmes sont dits *Turing-complets*, c'est-à-dire qu'ils calculent les mêmes choses que les machines de Turing (et que les λ-fonctions). En d'autres termes, cela signifie que tout problème qui peut être résolu par un algorithme peut l'être par une machine de Turing, par une fonction de Church ou par un programme écrit dans n'importe quel langage de programmation.

En effet, la portée universelle de la thèse de Church réside bien dans le fait que **la calculabilité ne dépend pas du langage de programmation**. C'est-à-dire qu'il n'existe aucune tâche qui puisse être réalisée par un programme dans un langage de programmation donné qui ne soit pas réalisable dans tous les autres.

Inversement, un problème qui ne peut pas être calculé par une machine de Turing ou par une fonction de Church, ne peut l'être par aucun programme, quel que soit le langage de programmation utilisé.

Décidabilité

Chacun de leur côté, Church et Turing ont explicité un problème dont ils ont montré que la solution n'était pas calculable.

Ces deux problèmes, que nous allons évoquer, sont tous deux des *problèmes de décision* dont voici la définition.

> Un **problème de décision** est un problème pour lequel la réponse est soit *oui* soit *non*.
>
> On appelle **décidable** un problème de décision dont la solution est calculable, et **indécidable** un problème de décision dont la solution n'est pas calculable.

Le problème l'arrêt (Turing)

Alan Turing s'est posé la question suivante, appelé **problème de l'arrêt** :

> Existe-t-il un algorithme (ou un programme) universel capable de déterminer, à partir d'un programme et d'une entrée, si ce programme s'arrête avec cette entrée ou non ?
>
> ***Le problème de l'arrêt***

Il a démontré qu'il ne pouvait exister de machine de Turing prenant en paramètres une machine $\boldsymbol{M}$ et une entrée $\boldsymbol{e}$, et déterminant si l'exécution de $\boldsymbol{M}$ sur l'entrée $\boldsymbol{e}$ termine un jour ou se poursuit infiniment. Autrement dit, il a prouvé que **le problème de l'arrêt est *indécidable***.

Une démonstration avec Python

On va expliquer la démonstration avec des outils plus modernes : des fonctions écrites dans le langage Python (on a vu que c'était équivalent à une machine de Turing ou à une fonction de Church ou à une fonction écrite dans n'importe quel autre langage).

> Cette démontration est basée sur un **raisonnement par l'absurde** qui permet de démontrer une proposition de la manière suivante : on suppose que cette proposition est *fausse*, puis, par une suite d'implications logiques, on aboutit à une affirmation *absurde* ou *contradictoire*. C'est que notre supposition de départ était incorrecte donc que la proposition n'est pas fausse, elle est donc vraie.

Supposons qu'il existe une fonction `arret` solution du problème de l'arrêt. Cette fonction prend une fonction `prog` et une entrée `x` en paramètres et renvoie `True` si `prog(x)` termine et `False` sinon :

```
def arret(prog, x):
    """Fonction qui renvoie Vrai si prog(x) termine, et False sinon."""
    ...  # inutile d'écrire un quelconque code car il ne peut pas en exister
```

Grâce à cette fonction, on va définir une fonction `paradoxe` de la manière suivante :

```
def paradoxe(prog):
    if arret(prog, prog):
        while True:
            pass
    else:
        return False
```

> Quel est le résultat de l'appel `paradoxe(paradoxe)` ?

Il n'y a que deux cas de figure :

- **1er cas** : l'appel `paradoxe(paradoxe)` rentre dans une boucle infinie si `arret(paradoxe, paradoxe)` est vrai donc si `paradoxe(paradoxe)` termine. Donc `paradoxe(paradoxe)` ne termine pas si `paradoxe(paradoxe)` termine...

- **2ème cas** : l'appel `paradoxe(paradoxe)` termine si `arret(paradoxe, paradoxe)` est faux donc si `paradoxe(paradoxe)` ne termine pas. Donc `paradoxe(paradoxe)` termine si `paradoxe(paradoxe)` ne termine pas...

Dans les deux cas on aboutit à une *contradiction*. Donc notre supposition de départ, à savoir qu'il existe une fonction `arret`, est fausse. On vient de démontrer qu'il ne peut pas exister une telle fonction, et donc que le problème de l'arrêt est indécidable.

> La démonstration proposée reprend l'idée qu'un programme est une donnée comme une autre qui peut être passée en paramètre à un autre programme. Ici, on a même appelé une fonction sur elle-même !

Le problème de l'égalité de deux fonctions (Church)

Alonzo Church s'est quant à lui posé la question suivante :

> Existe-t-il un algorithme (ou un programme) universel capable de déterminer si deux programmes font exactement la même chose sur les mêmes entrées.
>
> ***Le problème de l'égalité de deux fonctions***

Il a démontré, en s'appuyant sur les **λ**-fonctions, que ce **problème de l'égalité de fonctions est *indécidable***.

Revenons au problème de décision de Hilbert

En réalité, Turing et Church, ont montré respectivement que, comme les problèmes de l'arrêt et de l'égalité de deux fonctions sont indécidables, alors le problème de décision de Hilbert (*Entscheidungsproblem*) était aussi indécidable.

> Turing a raisonné par l'absurde : il a montré que s'il existait une machine de Turing qui résout le problème de décision, alors il existerait aussi une machine qui résout le problème de l'arrêt. Or, Turing a prouvé qu'il n'y en avait pas, donc il ne peut pas exister une machine qui résout le problème de décision. Et par universalité des machines de Turing, il ne peut exister aucune fonction calculable ou aucun algorithme qui résout l'*Entscheidungsproblem*.

Ainsi, Turing et Church ont chacun démontré, par des chemins bien différents, qu'il ne peut pas exister d'algorithme universel capable de décider si un énoncé mathématique est vrai ou faux. L'*Entscheidungsproblem* ne peut donc pas être résolu par un algorithme.

Indécidabilité : compléments

La notion de décidabilité ou d'indécidabilité algorithmique concerne des fonctions qui doivent répondre pour **toutes** les entrées possibles. Si un problème général est indécidable, cela ne veut pas dire que l'on ne peut pas y répondre dans des cas particuliers. Par exemple, il est possible de démontrer que la fonction suivante va s'arrêter sur toute entrée entière `n` positive :

```
def marrete_je(n):
    while n >= 0:
        print("Bonjour")
        n = n - 1
```

On peut utiliser pour cela ce qu'on appelle un **variant de boucle** (voir programme de Première).

De même, ce n'est pas parce que le problème de décision de Hilbert est indécidable que l'on ne peut pas prouver qu'un énoncé mathématique donné est vrai ou faux (il n'existe juste pas d'algorithme qui peut le faire pour *tous* les énoncés mathématiques).

Quelques années plus tard, en 1951, le logicien et mathématicien Henry Gordon Rice, a généralisé l'indécidabilité du théorème de l'arrêt avec ce qu'on appelle désormais le théorème de Rice qui énonce que « *toute propriété sémantique non triviale d'un programme est indécidable* ».

Une *propriété sémantique* concerne le comportement d'un programme (par opposition aux *propriétés syntaxiques* sur le code source, pour lesquels le théorème ne s'applique pas). Ainsi, d'après le théorème de Rice, il n'existe pas d'algorithme général qui répond à coup sûr aux questions suivantes :

- « le programme ne renvoie jamais la valeur 42 »
- « le programme ne termine jamais par une erreur d'exécution »

- « le programme calcule un résultat correct par rapport à sa spécification »
- « le programme contient un virus »

En particulier, on peut en déduire que :

- on ne peut pas savoir ce que fait un programme sans le faire tourner
- la preuve automatique de programmes est difficile
- il n'existe pas d'antivirus totalement efficace : un antivirus qui regarde si une signature est présente se contente de regarder les propriétés syntaxiques (« le code source contient-il une signature malveillante ? ») et donc le théorème de Rice n'est pas concerné. S'il regarde la sémantique (« ce que fait ce code est il dangereux ? »), il suffit d'appliquer le théorème de Rice pour voir que c'est impossible.

Bilan

- Un **problème de décision** est un problème pour lequel la réponse est soit *oui* soit *non*.
- Le problème de décision de Hilbert est l'un des événements fondateurs de l'informatique en tant que science : *existe-t-il un algorithme général capable de déterminer si un énoncé mathématique est vrai ou faux ?*
- Alonzo Church et Alan Turing ont tous les deux formalisé les notions de **fonctions calculables**, à savoir les fonctions qui peuvent être calculées par un algorithme. Cette théorie de la calculabilité permet ainsi de définir ce qu'un algorithme peut ou ne peut pas calculer.
- Leurs approches ont été très différentes : une vision mathématique pour Church avec les λ-fonctions et une vision informatique (mécanique) pour Turing avec ses machines de Turing.
- Leurs travaux se sont révélés équivalents : les fonctions pouvant être calculées selon la théorie du λ-calcul de Church sont exactement les mêmes que les fonctions qui peuvent être réalisées par une machine de Turing (on appelle cela la *thèse de Church-Turing*).
- Cette affirmation n'a jamais été contredite et tous les langages de programmation ordinaires peuvent calculer exactement les mêmes choses qu'une machine de Turing (ou que les fonctions λ-calculables de Church).
- Cela implique qu'un problème qui peut être résolu par une machine de Turing peut l'être avec n'importe quel langage de programmation. Inversement, si un problème ne peut pas être résolu par une machine de Turing, alors il ne peut l'être avec aucun langage de programmation. On dit que **la calculabilité ne dépend pas du langage de programmation**.
- Turing, avec le problème de l'arrêt, et Church, avec le problème de l'égalité de deux fonctions, ont tous les deux explicité un problème de décision dont ils ont montré qu'il était **indécidable**, c'est-à-dire qu'il n'existe pas de solution calculable au problème.
- C'est grâce à cela qu'ils ont peut déduire que le problème de décision de Hilbert était lui aussi indécidable : il n'existe pas d'algorithme universel capable de décider si un énoncé mathématique est vrai ou faux.

Références :

- Document ressource de l'équipe éducative du DIU EIL, Université de Nantes, CC BY : Les barrières de l'informatique (Bloc 5, Séance 5)
- Livre *Spécialité Numérique et sciences informatiques : 24 leçons avec exercices corrigés - Terminale*, éditions Ellipses, T. Balabonski, S. Conchon, J.-C. Filliâtre, K. Nguyen. Site du livre : [http://www.nsi-terminale.fr/]
- Livre *Prépabac, NSI, Terminale*, éditions Hatier, G. Connan, V. Petrov, G. Rozsavolgyi, L. Signac.
- Articles Wikipédia : Théorie de la calculabilité, Théorème de Rice
- Article Interstices : Comment fonctionne une machine de Turing ?

Germain BECKER, Lycée Mounier, ANGERS

Voir en ligne : info-mounier.fr/terminale_nsi/langages_prog/calculabilite-decidabilite