Calculabilité

III

Kévin Perrot – Aix Marseille Université – printemps 2016

Table des matières

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 38. *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 [1] 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 39. *Il existe des langages re qui ne sont pas récursifs, et la famille des langages récursifs n'est pas close par complémentation.*

1. 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.

4.1 Universalité

Le théorème 38 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) :

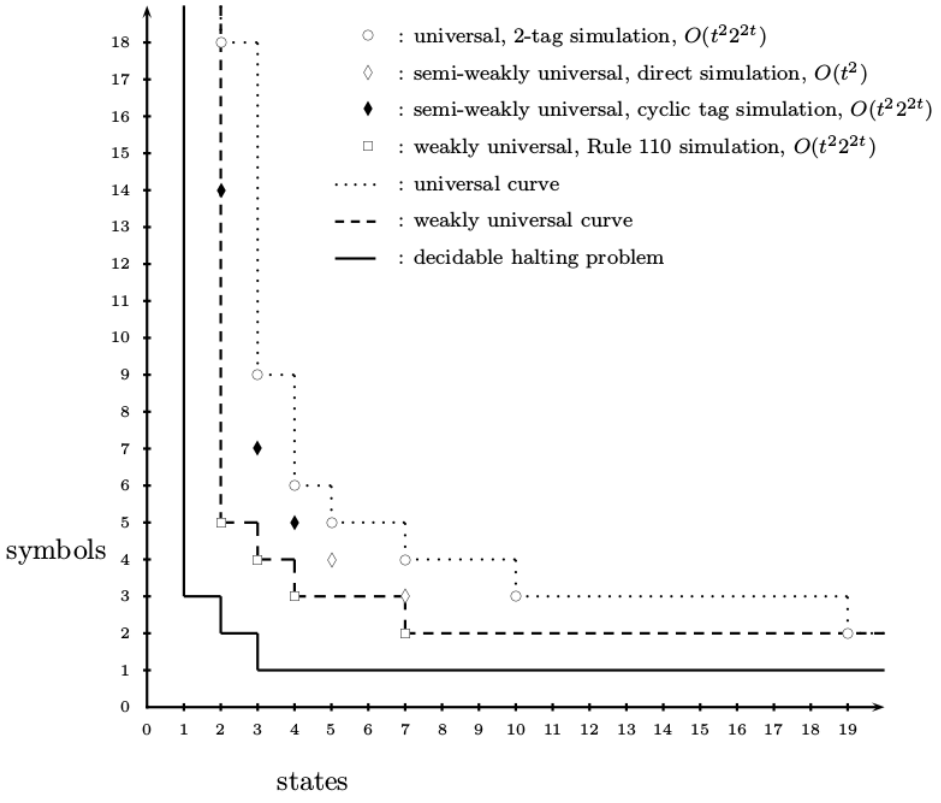
$$M_u(\langle M \rangle, w) = M(w).$$

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)
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 [4]. La figure suivante est issue de la thèse de doctorat de ce dernier, soutenue en 2008.



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). Exemples : les machines de Turing.

Définition 40. *Un modèle de calcul capable de calculer toutes les fonctions calculables par des machines de Turing est appelé* **Turing-complet**.

Remarque 41. *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 : vous pouvez implémenter un simulateur de machines de Turing.

Petite liste de modèles, jeux et langages Turing-complets (parfois accidentellement) :

— le λ-calcul (très minimaliste), 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$^{\text{TM}}$mécaniques (avec engrenages et pistons) : `http://www.dailymotion.com/video/xrmfie/`

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

Remarque 43. *Les MT non-déterministes et multi-tapes 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 [2]. En 1936, Alonzo Church définit le λ-calcul [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 !

2. 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.

3. En λ-calcul on définit des expression à base de termes x, de fonctions (λ abstraction) $\lambda x.x$, et d'application xy. Pour vous donner un aperçu, voici comment exprimer le nombre 2 : $\lambda s.\lambda z.s(sz)$; et l'addition a plus b : $\lambda a.\lambda b.\lambda s.\lambda z.as(bsz)$.

La thèse de Church-Turing, ou plutôt ses deux versions [5], 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 faire de calculer, ou de définir le calcul, reviendrait au même [4].

5.1 Thèse de Church-Turing version physique

Thèse 44. *Toute fonction physiquement calculable est calculable par une MT.*

Autrement dit tout modèle de machine, qui peut effectivement existe 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 [2].

Théorème 45. *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 [1] (par un binôme dont l'un est désormais chercheur à Aix-Marseille Université).

5.2 Thèse de Church-Turing version algorithmique

Thèse 46. *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 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.

6 Kleene et Quine

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

Il existe une *énumération des fonctions calculables*. Comme il est d'usage dans la littérature, nous noterons ϕ_e la $e^{\text{ième}}$ fonction calculable (nous avons vu qu'il existe une

4. Les machines quantiques n'échappent pas à la thèse de Church-Turing [1].

énumération des MT, ce qui revient au même que d'énumérer les fonctions calculables, puisque les fonctions calculables sont calculées par des MT !). Le nombre e peut être vu comme la représentation du code d'un programme.

Il existe un *interpréteur* (une machine de Turing universelle) $\phi_u(n, \dots) = \phi_n(\dots)$, qui prend en entrée la description d'une fonction calculable (le code d'une MT) et reproduit cette fonction (simule cette MT) sur le reste de l'entrée (les $\ll \dots \gg$).

Le **théorème s-m-n** est vrai : si une fonction ϕ_n est calculable, alors pour toute entrée x, il existe une fonction calculable $s(n, x)$ telle que $\phi_{s(n,x)}(\dots) = \phi_n(x, \dots)$.

6.1 Théorème du point fixe de Kleene

Théorème 47. *Pour toute fonction (totale) calculable h, il existe un programme n tel que*

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

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

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

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

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

6.2 Quine (*self-replicating programs*)

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

Définition 48. *Un* **quine** *est un programme qui affiche à l'écran son propre code source.*

Théorème 49. *Tout langage de programmation acceptable*[5] *admet des quines.*

Démonstration. Pour un programme t, considérons le programme $h(t)$ qui affiche à l'écran le code de t. Il est clair que la fonction h est calculable. Appliquons le théorème 47 à la fonction h : il existe un programme n tel que les programmes $h(n)$ et n sont identiques, donc n affiche à l'écran le code de n. □

Comment construire un tel programme en pratique ? Relisez les preuves, elles sont **constructives**, c'est-à-dire qu'elles prouvent l'existence de programmes possédant certaines propriétés tout en expliquant comment les construire.

5. Un langage de programmation est *acceptable* s'il vérifie les points précisés en début de section.

Un jeu d'initié[6] consiste à trouver le plus petit quine dans son langage préféré... Ci-dessous quelques exemples de Quine en C, Haskell, Java, OCaml, Python et en français (des sauts de lignes ont été ajoutés).

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

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

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

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

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

```
Recopier puis recopier entre guillemets la phrase
≪ Recopier puis recopier entre guillemets la phrase ≫
```

7 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 donné 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.

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

C'est la morale de l'histoire, qui rappelle (et ce n'est pas un hasard [7]) le premier théorème d'incomplétude de Gödel :

Théorème 50. *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.*

Références

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

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

[3] D. Madore. *personnal website*, consulté en avril 2016.

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

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

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