Calculabilité
Cours 4 : universalité et thèse de Church-Turing

Kévin Perrot – Aix Marseille Université – printemps 2019

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 1. *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 et seulement si M s'arrête dans son état final. □

Nous pouvons en déduire les corollaires suivants.

Théorème 2. *Il existe des langages re qui ne sont pas récursifs, et la famille des langages re 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.

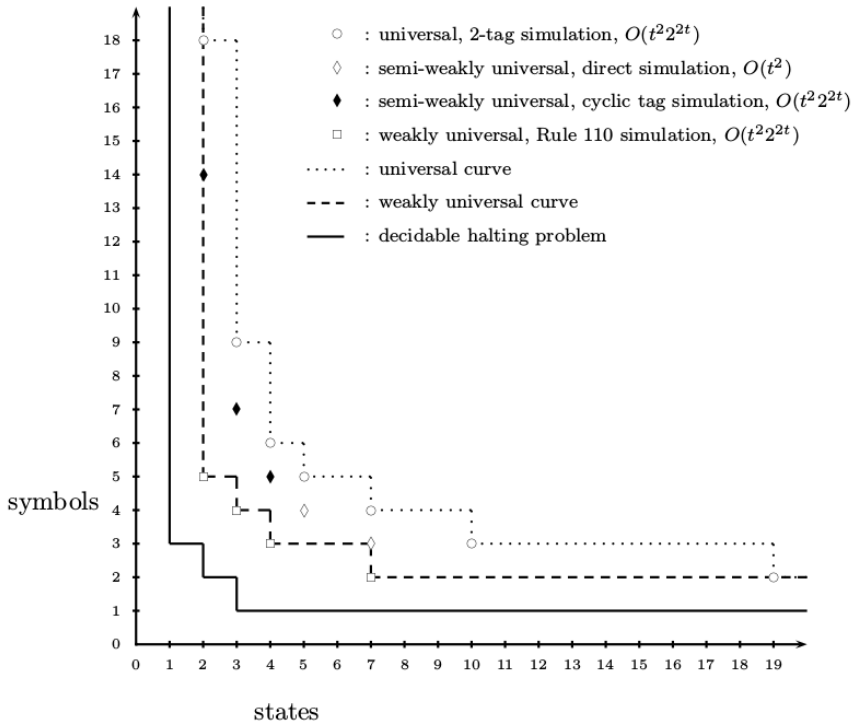

FIGURE 1 – Existence ou non de machines de Turing universelles pour un nombre d'états et de symboles de ruban donné.

4.1 Universalité

Le théorème 1 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 tout 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 [5]. La figure 1 est issue de la thèse de doctorat de ce dernier, soutenue en 2008.

Comment construire une machine de Turing universelle ?

Pour prouver leur existence, il suffit de donner un exemple de machine de Turing universelle. On pourrait le faire sans trop de problème à partir de notre encodage $\langle M \rangle$. Ce serait long à décrire entièrement, mais l'idée est simple : la machine universelle M_u prend en entrée $\langle M \rangle \# w$ avec $w = w_1 w_2 \dots w_n$ codé par

$$\underbrace{0\dots0}_{w_1}1\underbrace{0\dots0}_{w_2}1\dots1\underbrace{0\dots0}_{w_n}.$$

Pour simuler M sur l'entrée w, elle doit retenir l'état courant de M (en l'écrivant tout à gauche de son ruban par exemple, initiallement q_0 (encodé par 0) de la machine M) et la position courante (en marquant la case courant), et à chaque étape de M, la machine M_u doit :

1. chercher dans la liste des transitions de $\langle M \rangle$ si il y a une transition définie pour l'état courant et le symbole courant sous la tête de lecture (en comparant une à une avec toutes les transitions),
2. si une transition existe, alors changer l'état, le symbole et la position tel qu'indiqué dans cette transition, et recommender,
3. sinon s'arrêter, dans son état final ou non suivant si M s'arrête dans son état final ou non.

La difficulté principale tient au fait qu'une même machine M_u, qui a un nombre d'états et de symboles de ruban fixé, doit pouvoir simuler toute machine M, quels que soient son nombre d'états et de symboles de ruban. Pour cela M_u écrit ces informations en unaire sur le ruban, et compare en faisant des aller-retours. Ainsi M_u ne retient dans son état qu'une quantité finie d'information, et est bien une machine de Turing.

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

Remarque 4. *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),
— les fonctions μ-récursives (façon calculatoire de définir des fonctions),
— les pavages (un type de puzzles),
— les automates cellulaires (vit-on dans un AC ?),
— 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/`.

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

Remarque 6. *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 (détails en section 5.4). En 1936, Alonzo Church définit le λ-calcul (détails en section 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 [6], 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 [2].

5.1 Thèse de Church-Turing version physique

Thèse 7. *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 mieux 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 [3].

Théorème 8. *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),*

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

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 9. *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 appel 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.

Rappelons qu'Alan Turing est parti de l'idée d'un calculateur humain avec son stylo devant une feuille de papier, pour construire le modèle des machines qui portent son nom. Les machines de Turing sont-elles assez générales ? ou bien peut-on imaginer une autre façon non-équivalente de décrire les algorithmes, mais qui soit en accord avec notre intuition ? La Thèse 9 postule que les machines de Turing capturent en toute généralité la notion d'algorithme, et que l'on peut s'en contenter.

5.3 λ-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 (lambda abstraction), si t est un lambda terme et x une variable ;

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

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) [3].

$t[t'/x]$ 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.

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

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

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

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

On peut encoder l'algèbre de Boole :

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

(vérifier que $\texttt{nott} = \texttt{f}$ et $\texttt{notf} = \texttt{t}$ est un bon exercice) et alors voici la fonction qui retourne `t` (true) ou `f` (false) suivant si son entrée vaut 0 ou non : $\texttt{if0} = \lambda n.(n\ (\lambda x.\texttt{f})\ \texttt{t})$.

On peut créer des paires :

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

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 :

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

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

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

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

L'opérateur de point fixe `Y` partage la structure du terme $(\lambda x.x\ x)(\lambda x.x\ x)$ qui se réduit en lui-même (comme une fonction récursive qui se rappelle elle-même). On l'appelle également combinateur `Y`.

Équivalence avec les machines de 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 (une telle construction demande de bien comprendre l'opérateur de point fixe `Y`).

5.4 Fonctions μ-récursives

Le principe des fonctions μ-récursives est de décrire des fonctions algorithmiquement, c'est-à-dire avec une procédure indiquant comment les calculer [2].

On parle ici de définir les fonctions de $\mathbb{N}^*$ dans $\mathbb{N}$ qui sont « calculables ». Les fonctions sont construites à partir de fonctions de base, et d'opérateurs pour construire de nouvelles fonctions. Les fonctions primitives récursives sont une étape intermédiaire vers la définition des fonctions μ-récursives.

Fonctions primitives récursives

Les fonctions primitives récursives sont des fonctions de $\mathbb{N}^* \to \mathbb{N}$, définies inductivement à l'aide des fonctions de base suivantes :

— pour tous p et n, la fonction constante $f(x_1, \ldots, x_p) = n$,
— la fonction successeur, telle que $S(x) = x + 1$,
— pour tous p et $1 \leq i \leq p$, la i^{e} projection $P_i^p(x_1, \ldots, x_p) = x_i$,

et des opérateurs suivants :

— un opérateur de composition $\circ$ des fonctions, qui permet de construire

$$h \circ (g_1, \ldots, g_m) = f \text{ avec } f(x_1, \ldots, x_p) = h(g_1(x_1, \ldots, x_p), \ldots, g_m(x_1, \ldots, x_p)),$$

à partir d'une fonction h d'arité m, et de m fonctions $(g_i)_{1 \leq i \leq m}$ d'arité p,
— un opérateur de récursion primitive ρ, qui permet de construire

$$\begin{aligned} \rho(g, h) = f \quad \text{avec} \quad & f(0, x_1, \ldots, x_p) = g(x_1, \ldots, x_p) \\ \text{et} \quad & f(y+1, x_1, \ldots, x_p) = h(y, f(y, x_1, \ldots, x_p), x_1, \ldots, x_p), \end{aligned}$$

à partir d'une fonction g d'arité p, et d'une fonction h d'arité $p + 2$.

Formellement, l'ensemble des fonctions primitives récursives est défini comme le plus petit ensemble de fonctions contenant les fonctions de base et clos par les opérateurs $\circ$ et ρ.

L'intuition derrière l'opérateur de récursion primitive ρ est que :

— g est la condition initiale

— h est l'étape de récursion, dans laquelle on a accès à la valeur calculée précédemment, $f(y, x_1, \ldots, x_p)$.

Pour faire le lien avec les langages, on dit qu'un ensemble $A \subseteq \mathbb{N}^p$ est primitif récursif si et seulement si sa fonction caractéristique est primitive récursive. On note $\chi(A)$ la fonction caractéristique de A :

$$\chi(A)(x_1, \ldots, x_p) = \begin{cases} 1 & \text{si } (x_1, \ldots, x_p) \in A \\ 0 & \text{sinon.} \end{cases}$$

Exemples

L'addition est primitive récursive, avec

$$x + 0 = x$$
$$x + (y + 1) = (x + y) + 1$$

que l'on peut traduire en la fonction d'addition $ad : \mathbb{N}^2 \to \mathbb{N}$ avec

$$ad(x, 0) = P_1^1(x)$$
$$ad(x, y + 1) = S(P_3^3(x, y, ad(x, y))).$$

Selon la même idée la multiplication est aussi primitive récursive, avec

$$x \cdot 0 = 0$$
$$x \cdot (y + 1) = (x \cdot y) + x.$$

On peut également encoder l'idée d'un branchement (si alors sinon), avec

$$h(x_1, \ldots, x_p) = \begin{cases} f(x_1, \ldots, x_p) & \text{si } (x_1, \ldots, x_p) \in A \\ g(x_1, \ldots, x_p) & \text{sinon} \end{cases}$$

pour un langage primitive récursif A (ce qui signifie intuitivement que l'on peut tester, dans le cadre de la récursion primitive, si la condition est vérifiée ou non). Cela se traduit en

$$h = f \cdot \chi(A) + g \cdot \chi(\mathbb{N}^p \setminus A).$$

On pourrait continuer loin, par exemple le test de primalité est primitif récursif.

Ackermann

On peut définir de nombreuses fonctions grâce à la récursion primitive. Cependant, il semble leur manquer quelque chose (qui sera comblé par l'ajout de l'opérateur μ qui donnera les fonction μ-récursives), car il existe des fonctions qui semblent intuitivement calculables, mais qui ne sont pas récursives primitives. Un tel exemple est la fonction d'Ackermann, que nous noterons $\xi : \mathbb{N}^2 \to \mathbb{N}$, et qui est définie comme suit.

$$\text{pour tout } x \in \mathbb{N}, \xi(0, x) = 2^x$$
$$\text{pour tout } y \in \mathbb{N}, \xi(y, 0) = 1$$
$$\text{pour tout } x, y \in \mathbb{N}, \xi(y + 1, x + 1) = \xi(y, \xi(y + 1, x))$$

On peut vérifier que ξ est bien définie et que le dépliage de la récursion termine toujours (en considérant l'ordre lexicographique sur (y, x) on voit que les couples décroissent à chaque appel récursif).

Il est possible de démontrer que la fonction d'Ackermann ξ croît plus vite que toute fonction primitive récursive, donc elle n'est pas récursive primitive.

Fonctions μ-récursives

Les fonctions μ-récursives sont obtenues en ajoutant l'opérateur :
— un opérateur de minimisation μ, qui permet de construire

$$\mu(f)(x_1, \ldots, x_p) = z \iff \begin{array}{l} f(z, x_1, \ldots, x_p) = 0 \text{ and} \\ f(i, x_1, \ldots, x_p) > 0 \text{ pour tout } i \in \{0, \ldots, z-1\}, \end{array}$$

à partir d'une fonction f d'arité $p + 1$.
et en prenant encore une fois le plus petit ensemble de fonctions contenant les fonctions de base et clos par les opérateurs $\circ$, ρ et μ.

L'intuition derrière l'opérateur de minimisation μ est de rechercher, en partant de 0 et en augmentant, le plus petit z tel que f retourne 0. Si un tel argument n'existe pas, alors la recherche ne termine pas.

Équivalence avec les machines de Turing

Toute fonction μ-récursive est calculable par une machine de Turing, et toute fonction calculable par une machine de Turing est μ-récursive. C'est technique, mais on peut effectivement convertir une définition de fonction μ-récursive, en une machine de Turing qui calcule la même fonction, et inversement.

On retrouve nos ensemble récursifs et récursivement énumérable de la façon suivante.

Définition 10. *Un langage $A \subseteq \mathbb{N}^p$ est récursif ssi sa fonction caractéristique $\chi(A)$ est μ-récursive et totale*[4]*. Un langage $A \subseteq \mathbb{N}^p$ est récursivement énumérable ssi c'est le domaine de définition*[5] *d'une fonction partielle μ-récursive.*

5.5 Automates cellulaires : le jeu de la vie

Un *automate cellulaire* en dimension d sur l'alphabet fini A est une fonction $F : A^{\mathbb{Z}^d} \to A^{\mathbb{Z}^d}$ définie par un voisinage fini $U \subset \mathbb{Z}^d$ et une fonction locale $f : A^U \to A$ par

$$\forall i \in \mathbb{Z}^d : F(x)(i) = f(x_{i+U}),$$

où x_{i+U} est la fonction $j \in U \mapsto x(i+j)$. Un automate cellulaire est un système dynamique discret (en espace et en temps) sur l'ensemble de configurations $A^{\mathbb{Z}^d}$ (une configuration associe à chaque cellule de $\mathbb{Z}^d$ une lettre de A. Un automate cellulaire est défini par un triplet (A, U, f) (la dynamique est définie par la fonction locale f qui permet de calculer l'image de chaque cellule par F).

Le *jeu de la vie* (*game of life*) est un exemple très fameux d'automate cellulaire, défini par Conway dans les années 1970. Il est défini en deux dimensions sur l'alphabet $\{0, 1\}$

4. c'est-à-dire définie sur tout $\mathbb{N}^p$.
5. en considérant que si une valeur est retournée, c'est 1.

avec le voisinage de Moore (9 voisins) $U = \bigcup\{(i,j) \mid i,j \in \{-1,0,1\} \text{ et } |i|+|j| \leq 2\}$ par la fonction locale

$$f : x_U \mapsto \begin{cases} 1 \text{ si } x_{(0,0)} = 0 \text{ et } \sum\limits_{i \in U} x_i = 3 \\ 1 \text{ si } x_{(0,0)} = 1 \text{ et } 3 \leq \sum\limits_{i \in U} x_i \leq 4 \\ 0 \text{ sinon.} \end{cases}$$

Avec des mots, on dit qu'une cellule est *vivante* si son état est 1 et *morte* si son état est 0. Alors une cellule morte peu naître si elle a exactement 3 voisines vivantes (il faut 3 parents pour naître), et une cellule vivante reste vivante si elle a entre 2 et 3 voisines vivantes (trop peu et elle meurt d'isolation, trop et elle meurt de surpopulation). Une configuration est *finie* si elle contient un nombre fini de cellules vivantes, et on appelle *motif* la projection/restriction d'une configuration sur un support fini.

On peut voir le jeu de la vie comme un modèle de calcul, et considérer les problèmes :

— étant donnés deux motifs A et B, le motif B va-t-il apparaître au cours de l'évolution depuis A ?

— étant donnée une configuration initiale finie, toutes les cellules vont-elles mourir ?

— étant donnée une configuration initiale finie et une cellule morte, cette cellule va-t-elle naître au cours de l'évolution ?

qui sont tous indécidables.

Pour jouer :

— simulateur du jeu de la vie : `http://golly.sourceforge.net/`,

— bibliothèque de motifs : `http://conwaylife.com/wiki/Main_Page`.

Références

[1] P. Arrighi and G. Dowek. The physical Church-Turing thesis and the principles of quantum theory. *Int. J. of Found. of C.S.*, 23 :1131–1145, 2012. arXiv:1102.1612.

[2] R. Cori and D. Lascar. *Logique mathématique 2 - Fonctions récursives, théorème de Gödel, théorie des ensembles, théorie des modèles.* Dunod, 2003.

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

[4] J. Garrigues. Cours : computability and lambda calculus. 2013.

[5] T. Neary and D. Woods. The complexity of small universal turing machines. In *proceedings of CiE'2007, volume 4497 of LNCS*, pages 791–798, 2007. arXiv:1110.2230.

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