Polycopié de calculabilité

Nicolas Ollinger

Avant-Propos Nous proposons au lecteur quelques éléments d'introduction à la théorie de la calculabilité autour de modèles de calcul classiques. Il s'agit d'une invitation à poursuivre par la lecture de références plus complètes comme les manuels d'O. Carton [3] ou de M. Sipser [34] voire à prolonger par des références plus exhaustives comme P. Odifreddi [24].

1 Comment bien compter ses moutons

Il était une fois, en des temps lointains, une bergère qui gardait ses moutons. Afin de s'assurer que tous les animaux qui sortent de la bergerie le matin réintègrent le bâtiment à la tombée de la nuit, la bergère se munit d'une jarre et d'une collection de galets. Pour chaque mouton qui quitte la bergerie, elle ajoute un galet à la jarre. Pour chaque mouton qui rentre à la bergerie, elle soustrait un galet à la jarre. À la fin de la journée, si la jarre est vide c'est que tous les moutons ont réintégré leur enclos.

Cette parabole du berger et ses variantes[1] sont souvent utilisées pour illustrer le mot *calcul* qui vient du latin *calculus* : petite pierre, caillou. Il s'agit ici de compter sans nombres en mettant en bijection deux ensembles. Cette parabole symbolise aussi le début d'une course scientifique et technologique, l'invention et le perfectionnement d'outils de calcul de plus

en plus puissants pour assister l'être humain à réaliser de plus en plus rapidement de grosses quantités de calculs de plus en plus complexes.

Quelques siècles plus tard, l'ordinateur sur lequel je compose ce texte est le descendant de cette jarre de galets : une machine à calculer automatique, polyvalente, à programme stocké en mémoire et dont le fabricant met en avant les 16 000 000 000 transistors du circuit intégré principal (système sur une puce intégrant processeur, mémoire, coprocesseurs divers, ...) et son horloge cadencée à plus de 3 GHz. Cette machine code l'information en binaire, avec des 0 et des 1, permettant la représentation de n'importe quelle donnée discrète sous forme numérique. Les transistors sont combinés pour réaliser des portes logiques et des bascules pour mettre en œuvre des circuits booléens séquentiels en faisant abstraction de la technologie du dispositif physique utilisé (les tubes à vide des années 40 ont cédé la place à la technologie CMOS). Des supports physiques et des réseaux numériques permettent de stocker et de transmettre cette information numérique d'ordinateur en ordinateur.

Les circuits booléens, fixes, mis en œuvre sur le circuit intégré de cet ordinateur sont structurés selon des grands principes d'organisation qui dérivent en droite ligne de l'*architecture de von Neumann*. Ce modèle d'organisation pour un calculateur automatique polyvalent à programme stocké en mémoire est décrit par J. von Neumann en 1945 dans le remarquable *First Draft of a Report on the EDVAC* [38]. Il organise un ordinateur en plusieurs composants : une unité arithmétique et logique dédiée au calcul ; une unité de contrôle en charge de la coordination des opérations ; des mémoires stockant données et programmes ; des dispositifs d'entrées-sorties échangeant avec l'extérieur. Une contribution majeure de cette architecture est de stocker le programme directement en mémoire, contrairement aux calculateurs de la génération précédente comme l'ENIAC qui étaient programmables en modifiant physiquement le câblage de la machine. Aujourd'hui, il est devenu banal d'éditer un programme sur un ordinateur et de l'exécuter sur le même ordinateur dans la foulée. Mis à part les limites imposées par la quantité de mémoire de la machine et sa vitesse de calcul, les ordinateurs contemporains, polyvalents, sont tous en capacité d'exécuter les mêmes programmes. Les calculateurs classiques ne se distinguent plus par l'expressivité des opérations qu'ils sont capables d'effectuer.

La bergère de l'histoire serait sans doute surprise d'apprendre qu'avec trois jarres au lieu d'une, une collection conséquente de cailloux, des règles un peu plus sophistiquées à appliquer et beaucoup de temps libre, elle serait en mesure de calculer tout ce que calcule mon ordinateur moderne[2]. Il s'agit d'une contribution essentielle au succès des ordinateurs contemporains :

l'unification de tout ce qui est calculable sur le discret, capturé par de nombreux modèles de calcul, tous équivalents, universels pour le calcul. Cette grande théorie unificatrice, qui pose aussi les limites de ce qui est calculable et de ce qui ne l'est pas, est née dans le creuset de la crise des fondements des mathématiques au début du XX^{e} siècle. Le modèle de calcul universel de la machine de Turing, les premiers problèmes démontrés indécidables et l'existence d'une machine universelle parmi ces machines sont présentés par A. Turing en 1936 dans le remarquable *On computable numbers with an application to the Entscheidungs problem* [36]. L'architecture de von Neumann constitue en quelque sorte la réalisation physique de la machine universelle de Turing[3].

B. Pascal, G. Leibniz, J. de Vaucanson, J.-M. Jacquard, Ch. Babbage, A. Lovelace, C. Shannon, ... La liste est est longue des personnes ayant apporté une contribution pour aboutir à l'invention de l'ordinateur universel et nous laisserons aux historiens la difficile tâche d'identifier le rôle de chacun. Le lecteur qui voudrait en apprendre davantage sur le cheminement des idées scientifiques et technologiques depuis la mise en lumière de l'arithmétique binaire par G. Leibniz et son rêve d'un calculateur symbolique, jusqu'à l'apparition des calculateurs numériques modernes, est invité à consulter l'ouvrage que M. Davis [8] consacre au sujet. Concernant l'histoire de la théorie de la calculabilité, le lecteur est invité à lire le texte de R. Soare [35].

La suite de ce chapitre propose un tour d'horizon de la théorie de la calculabilité en empruntant un cheminement proche de celui de M. Minsky dans son atypique *Computation : Finite and Infinite Machines* [22]. Après avoir établi un lien entre circuits booléens séquentiels et automates finis de Mealy, nous discutons de l'ajout de différentes sortes de mémoires non bornées à un tel automate et du calcul avec de telles machines infinies. La thèse de Church-Turing permet ensuite de nous abstraire du choix d'un modèle particulier du fait de son universalité. Nous abordons les machines universelles au sein de leur modèle qui permettent de confondre données et programmes, la machine universelle devenant le modèle de machine à calculer polyvalente. Enfin, nous esquissons les premiers résultats d'indécidabilité et les très utiles réductions entre problèmes de décision. Ce chapitre n'aborde pas les questions d'efficacité du calcul qui sont l'objet de la théorie de la complexité[4].

2 Machines finies et circuits booléens

Les règles déterministes qui régissent le comportement de la bergère, ses décisions d'ajouter ou de supprimer des galets, à partir des observations de départs et d'arrivées de moutons, nécessitent de mémoriser une quantité finie d'information. Il en va de même pour un circuit intégré qui calcule les valeurs des bits sur ses broches de sortie à partir des valeurs des bits sur ses broches d'entrée et de l'état interne dans lequel se trouve le circuit intégré. Les automates de Mealy constituent un modèle adapté pour décrire ce type de contrôle fini et ces automates caractérisent précisément ce qui est calculable par des circuits booléens séquentiels.

Remarquons tout d'abord que tout ensemble fini peut être codé sur un nombre suffisant de bits, quitte à ne pas utiliser certaines valeurs. Il en résulte que toute fonction $f : A \to B$ définie sur des ensembles finis A et B peut être assimilée à une fonction binaire $f : \{0,1\}^m \to \{0,1\}^n$ pour des valeurs appropriées de m et n. Cette fonction peut être décomposée en n fonctions booléennes $f_i : \{0,1\}^m \to \{0,1\}$ de sorte que $f(x_1,\dots,x_m) = (f_1(x_1,\dots,x_m),\dots,f_n(x_1,\dots,x_m))$. On retrouve le cadre de l'algébre booléenne en identifiant 0 à *faux* et 1 à *vrai*.

Un *circuit combinatoire* est un DAG (graphe simple orienté sans cycle) dont les sources sont les entrées du circuit, les puits sont les sorties et dont les autres sommets portent des portes logiques, des fonctions booléennes, de sorte que le degré entrant de chaque sommet respecte l'arité de la porte associée. Tout circuit combinatoire calcule une fonction binaire associée, obtenue en évaluant le circuit des entrées vers les sorties (formellement selon un tri topologique). À titre d'exemple, la figure 1 représente un circuit combinatoire comportant uniquement des portes NON-ET (de fonction booléenne $x \uparrow y = 1 - xy$) et qui calcule le résultat de l'addition de trois bits (la sortie comporte donc deux bits).

Théorème 2.1. *Toute fonction binaire est calculée par un circuit combinatoire comportant uniquement des portes NON-ET.*

Démonstration. On montre d'abord par récurrence sur n que toute fonction booléenne $f : \{0,1\}^n \to \{0,1\}$ est calculable par un circuit combinatoire comportant des portes ET, OU et NON, en remarquant que $f(x_1,\dots,x_n) = (1-x_n)\cdot f(x_1,\dots,x_{n-1},0) + x_n \cdot f(x_1,\dots,x_{n-1},1)$. Puis on réalise ces trois portes avec des portes NON-ET grâce aux identitées $1 - x = x \uparrow x$, $xy = (x \uparrow y) \uparrow (x \uparrow y)$ et $\max(x,y) = (x \uparrow x) \uparrow (y \uparrow y)$. ■

Les circuits combinatoires peuvent calculer toute sorte de choses, par

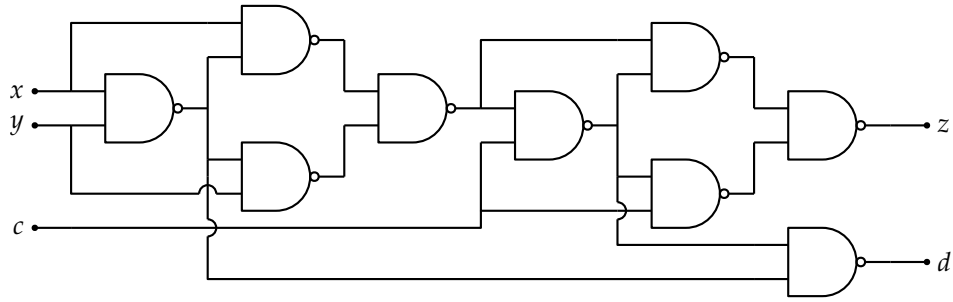


FIGURE 1 – *Addition de trois bits avec neuf portes NON-ET.*
On vérifie par le calcul que $z = x + y + c \pmod 2$ *et* $d = \left\lfloor \frac{x+y+c}{2} \right\rfloor$.

exemple additionner des nombres bornés exprimés sur un nombre fixé de bits. En revanche, ils ne permettent pas de mémoriser un état.

Des phénomènes intéressants apparaissent dans les circuits logiques lorsqu'on ajoute des cycles comme sur la figure 2. Les valeurs observées en sorties des circuits ne dépendent plus uniquement des valeurs en entrée mais aussi des sorties précédentes : $y(t+1) = F(x(t), y(t))$. Les bascules/verrous comme ceux de la figure 2 permettent de créer des registres mémoire : des circuits qui mémorisent des bits et permettent de mettre à jour le contenu de cette mémoire.

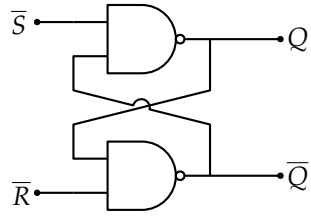


FIGURE 2 – *Exemple de circuit séquentiel non combinatoire : verrou* $\overline{RS}$

Ces mémoires permettent de construire des *circuits séquentiels* (synchrones) : l'état interne est mémorisé dans un registre et à chaque itération, l'entrée et l'état interne traversent un circuit combinatoire pour calculer le nouvel état interne et la sortie, comme illustré sur la figure 3.

Un *automate de Mealy* [21] est la donnée d'une fonction locale de transition $\delta : Q \times I \to Q \times O$ et d'un état initial q_0. Pour tout mot $u \in I^*$ sur l'alphabet d'entrée, partant de l'état q_0, l'automate produit un mot $v \in O^*$ de même longueur sur l'alphabet de sortie en appliquant séquentiellement

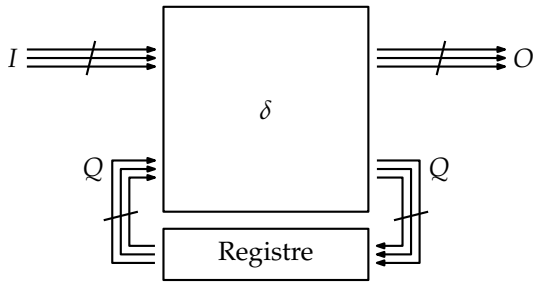


FIGURE 3 – *Principe de fonctionnement d'un circuit séquentiel*

la fonction de transition : pour tout $0 < i \leqslant n$ on pose $(q_i, v_i) = \delta(q_{i-1}, u_i)$ où $u = u_1 \cdots u_n$ et $v = v_1 \cdots v_n$.

Théorème 2.2. *Tout automate de Mealy peut être mis en œuvre par un circuit séquentiel synchrone composé de portes NON-ET et de registres mémoire.*

À partir de cette caractérisation, il est possible de concevoir toute une architecture à partir uniquement de portes NON-ET et de verrous[5], c'est le domaine de l'architecture des ordinateurs [26]. Les automates de Mealy participent quand à eux à l'histoire de la théorie des automates[6] [28].

Les automates de Mealy constituent un modèle de machine finie qui peut être couplée à l'environnement extérieur par le biais des entrées et des sorties. L'expérience la plus simple consiste à utiliser ces automates pour calculer des fonctions $f : I^* \to O^*$ où I et O sont des alphabets finis. On obtient ainsi une première notion de fonction calculable par automate.

À titre d'exemple, la figure 4 additionne deux entiers codés, bit de poids faible en tête, en binaire sur un même nombre de bits, $x_1 \cdots x_n$ et $y_1 \cdots y_n$. L'automate reçoit la séquence de paires de bits (x_1, y_1), (x_2, y_2), $\ldots$, (x_n, y_n), $(0, 0)$ et produit une séquence de bits $z_1, \ldots, z_{n+1}$ qui code leur somme, bit de poids faible en tête. L'état interne de l'automate est utilisé pour mémoriser la retenue.

Le modèle des machines finies est cependant très limité dans son pouvoir de calcul et des fonctions aussi simples que la multiplication de deux entiers quelconques est hors de portée de ce modèle de calcul.

Proposition 2.3. *La multiplication de deux entiers codés en binaire, bit de poids faible en tête, n'est pas calculable par automate de Mealy.*

Démonstration. La démonstration procède par un argument classique de pompage, typique sur les automates finis. Comme $2^n \times 2^n = 2^{2n}$ pour tout $n \in \mathbb{N}$, un automate de Mealy qui calcule le produit doit associer la

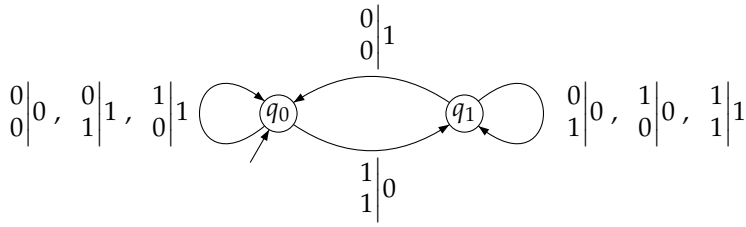


FIGURE 4 – *Addition de deux nombres binaires de longueur non bornée par un automate de Mealy à deux états, d'alphabet d'entrée* $\{0,1\}^2$ *et d'alphabet de sortie* $\{0,1\}$. *La notation* $\frac{x_i}{y_i}\Big|z_i$ *signifie que l'automate lit la paire* (x_i, y_i) *et produit* z_i.

sortie $\underbrace{0\cdots0}_{2n}1$ à l'entrée $\underbrace{(0,0)\cdots(0,0)}_{n}(1,1)\underbrace{(0,0)\cdots(0,0)}_{n}$. En particulier, sur une plage de longueur arbitraire d'entrée $(0,0)\cdots(0,0)$, il produit 0, avant finalement de produire 1 sur la même entrée. Un tel changement de comportement n'est pas possible sur une plage plus longue que le nombre d'états de l'automate car l'automate est déterministe. ■

Intuitivement, ce résultat n'est pas surprenant. Lorsqu'on pose une multiplication à la main, le calcul nécessite d'effectuer plusieurs additions successives, de manipuler des retenues, etc. Il ne semble pas raisonnable de demander à ce que pour tout calcul la sortie soit produite immédiatement. Un couplage plus fin de la machine finie à son environnement résout ce problème, c'est l'idée formalisée par A. Turing et qui est décrite plus loin.

Puisque nos ordinateurs physiques calculent grâce à des circuits intégrés qui réalisent des circuits séquentiels fixes, il est tentant d'utiliser le modèle des automates de Mealy pour en décrire les capacités de calcul. Si ce modèle est adapté pour la conception des circuits et la mise au point de la partie contrôle, c'est un très mauvais modèle pour décrire l'ensemble des algorithmes. De la même manière que l'abstraction des entiers naturels, en nombre non borné, est une meilleure abstraction que de se retreindre à la collection finie des nombres inférieurs au nombre d'atomes de l'univers, les machines à mémoire non bornée nous fournissent un meilleur modèle pour décrire nos algorithmes. Ces algorithmes sont ensuite traduits dans des langages de programmation et compilés vers des machines à mémoire finies qui les exécutent sur une portion finie des entrées possibles. La plupart du temps, en pratique, c'est le temps de calcul qui posera problème plutôt que l'espace mémoire disponible.

3 Machines à mémoire non bornée

Au début du XX^{e} siècle, les mathématiques sont en pleine crise des fondements [11] : D. Hilbert est invité à identifier les problèmes les plus importants pour le nouveau siècle, les fameux problèmes de Hilbert, et liste parmi ceux-ci le programme de refonder proprement l'arithmétique grâce à la logique. Il s'agit aussi de trouver des méthodes, *des algorithmes*, pour permettre un traitement automatique des théories et énoncés logiques, leurs démonstrations et leur valeur de vérité. En 1931, les théorèmes d'incomplétude de K. Gödel ont marqué un premier coup d'arrêt à ce programme d'automatisation. À peine, en 1928, D. Hilbert et W. Ackermann ont posé l'*Entscheidungsproblem* qu'en 1936, A. Church [4] et A. Turing [36] apportent une réponse inattendue qui termine d'enterrer ce projet : à la question « *définir une méthode effective* » ils répondent « *une telle méthode n'existe pas* ».

L'*Entscheidungsproblem*, ou problème de la décision, demande de définir une méthode effective pour décider si un énoncé de la logique du premier ordre est universellement valide. D'après le théorème de complétude de K. Gödel, cela revient à décider si l'énoncé possède une démonstration dans le calcul des prédicats. Dans le cas où un tel énoncé est valide, il suffit d'énumérer toutes les démonstrations jusqu'à exhiber une démonstration qui correspond à l'énoncé. Mais comment traiter le cas où l'énoncé n'est pas valide ?

Pour démontrer formellement qu'une telle méthode n'existe pas, qu'il n'existe pas d'algorithme, il faut au préalable définir précisément ces méthodes effectives. Pour cela, A. Turing s'inspire de l'activité d'un mathématicien en train de calculer avec des nombres et transpose cette activité en le comportement d'une machine.

Un mathématicien qui calcule est comme la bergère en train de compter ses moutons : il n'est capable de mémoriser qu'un nombre fini de conditions valides ou non à chaque instant de son calcul. Si sa méthode de calcul est déterministe, il pourra donc avantageusement être remplacé par une machine finie, un automate qui effectue le calcul. Pour cela, il dispose d'un support, typiquement des feuilles de papier ordonnées sur lesquelles il lit et écrit des symboles durant son calcul. On suppose que l'encre et le papier sont disponibles en quantité illimitées. La quantité d'information que peut contenir une feuille est bornée et quitte à considérer un alphabet très grand, on peut assimiler le contenu des feuilles à un unique symbole. Au départ les feuilles sont blanches, à l'exception de quelques feuilles qui portent les données initiales du calcul. Pendant qu'il calcule, le mathématicien peut passer d'une feuille à une autre en maintenant l'ensemble des feuilles

ordonnées. Plutôt qu'un énorme cahier, on peut simplifier la maintenance de cette mémoire de travail en la représentant par un ruban de papier biinfini découpé en cases, chaque case représentant une feuille et contenant donc un symbole. L'automate travaille à chaque instant sur une case du ruban et peut se déplacer le long du ruban pendant son calcul. Une fois que l'automate déclare le calcul terminé, on peut lire le résultat du calcul sur le ruban à partir de la case courante et jusqu'à la prochaine case blanche.

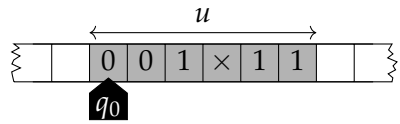
À quelques détails syntaxiques près, dont nous verrons qu'ils importent peu car le modèle est très robuste, c'est ainsi qu'A. Turing formalise les machines qui portent son nom.

3.1 Machines de Turing

Définition 3.1. *Une* machine de Turing *est un tuple* $(Q, \Gamma, \Sigma, \delta, q_0, B, q_F)$ *où*
- Q *est l'ensemble fini des états ;*
- Γ *est l'alphabet fini de travail ;*
- $\Sigma \subseteq \Gamma$ *est l'alphabet d'entrée ;*
- $\delta : Q \times \Gamma \rightharpoonup Q \times \Gamma \times \{\blacktriangleleft, \blacktriangledown, \blacktriangleright\}$ *est la fonction de transition, partielle ;*
- $q_0 \in Q$ *est l'état inital ;*
- $B \in \Gamma \setminus \Sigma$ *est le symbole blanc ;*
- $q_F \in Q$ *est l'état d'acceptation de la machine.*

Une fois posée cette définition très syntaxique, il convient d'expliquer la dynamique d'une telle machine $\mathcal{M}$ et comment l'utiliser afin de calculer sa fonction partielle $f_{\mathcal{M}} : \Sigma^* \rightharpoonup (\Gamma \setminus \{B\})^*$.

Pour calculer à partir d'un mot d'entrée $u \in \Sigma^*$, on place la machine dans la *configuration initiale* $c_0(u)$ associée : le ruban est partout blanc (les cases contiennent le symbole B) sauf sur une portion finie qui porte la suite des symboles $u_1 \cdots u_n$ de l'entrée ; la tête de lecture/écriture pointe sur la première lettre de l'entrée et la machine est dans l'état q_0.



Plutôt que de voir l'espace des configurations comme un élément de $Q \times \Gamma^{\mathbb{Z}} \times \mathbb{Z}$, on se contentera ici de décrire une configuration par une *description instantanée* notée $uqv \in \Gamma^* \times Q \times \Gamma^*$ où uv représente une portion du ruban qui contient tous les symboles non blancs, la machine est dans l'état q et la tête de lecture/écriture pointe sur la première des cases de la portion du ruban qui contient v. Avec cette notation $c_0(u) = q_0 u$.

Définition 3.2. *Une* configuration *est un triplet $uqv \in \Gamma^* \times Q \times \Gamma^*$. L'espace des configurations est quotienté par l'ajout de symboles B à gauche de u ou à droite de v, i.e. les configurations $Buqv$, $uqvB$ et uqv sont considérées comme une même et unique configuration.*

Le calcul de la machine sur l'entrée $u \in \Sigma^*$ est l'enchaînement déterministe de configurations $c_0(u) \vdash c_1 \vdash \cdots \vdash c_i \vdash c_{i+1} \vdash \cdots$ qui débute par la configuration initiale et effectue des *pas de calcul*, des *transitions* successives, déterminées par la fonction de transition. Une *règle de transition* $\delta(q, a) = (q', b, \Delta)$ se lit : « *dans l'état q, lorsqu'elle lit le symbole a, la machine passe dans l'état q', écrit le symbole b et se déplace dans la direction* Δ ». La figure 5 illustre le déroulement d'un calcul.

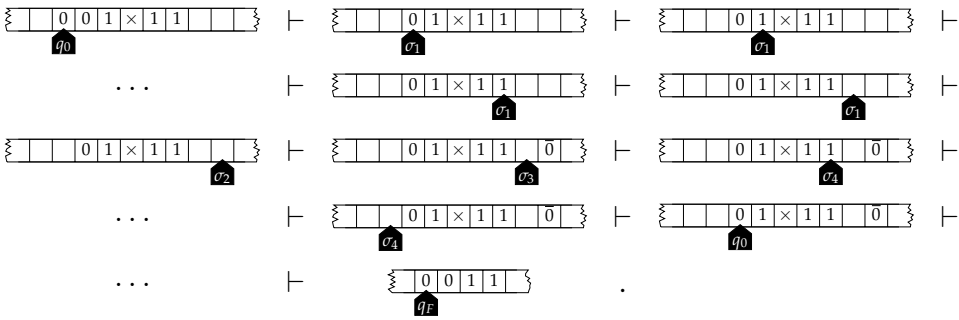


FIGURE 5 – *Exemple de calcul d'une machine de Turing.*

Définition 3.3. *Un* pas de calcul $c \vdash c'$ *transforme une configuration c en une configuration c' selon les règles suivantes :*

$$\begin{aligned} ua'qav &\vdash uq'a'bv && \text{si } \delta(q,a) = (q', b, \blacktriangleleft) \\ uqav &\vdash ubq'v && \text{si } \delta(q,a) = (q', b, \blacktriangleright) \\ uqav &\vdash uq'bv && \text{si } \delta(q,a) = (q', b, \blacktriangledown) \end{aligned}$$

pour tous $a, a', b \in \Gamma$, $u, v \in \Gamma^*$, $q, q' \in Q$. On note $\vdash^k$ l'itération, $\vdash^+$ la clôture transitive et $\vdash^*$ la clôture réflexo-transitive de la relation $\vdash$ sur les configurations.

Une *configuration terminale* est une configuration pour laquelle aucun pas de calcul n'est possible. Un calcul sur une entrée u peut être fini, on dit alors que la machine *s'arrête* sur u. Si la configuration terminale qu'elle atteint a pour état q_F, on dit que la machine *accepte* l'entrée u et on lit la sortie sur cette configuration terminale $v'q_Fv$ comme le plus long préfixe v_0

de v qui ne contient pas B, on pose $f_{\mathcal{M}}(u) = v_0$. La machine *rejette* le mot u si elle termine dans un état différent de q_F ou si le calcul est infini, dans ce dernier cas on dit aussi que la machine *diverge* sur l'entrée u. La fonction $f_{\mathcal{M}}$ n'est pas définie sur les mots que la machine rejette. La machine est *totale* si elle s'arrête sur toute entrée.

Définition 3.4. *Une fonction partielle $f : \Sigma^* \rightharpoonup \Gamma^*$ est Turing-calculable si elle est calculée par une machine de Turing.*

À titre d'exemple, la machine de la figure 6 calcule le produit de deux entiers codés en binaire. On notera que la machine est décrite sous forme graphique pour faciliter sa compréhension. On notera aussi que les machines de Turing acceptent une entrée unique, aussi pour communiquer les deux entiers à multiplier on a recours à un codage. En fin de chapitre[7], le lecteur intéressé trouvera un lien vers un simulateur de machines de Turing pour expérimenter avec ce modèle.

La définition de fonctions Turing-calculables peut sembler a priori très spécifique. Il n'en est rien comme nous le verrons plus loin : cette notion capture une idée universelle de procédure effective de calcul. Avant d'aborder ce point, nous faisons un détour par une autre manière d'utiliser les machines de Turing, pour reconnaître des langages.

Remarque 3.5. *Une machine de Turing est donc bien constituée d'un automate de Mealy couplé à un ruban infini muni d'une tête de lecture/écriture qui se déplace le long du ruban. La machine communique avec le ruban à travers les alphabets finis d'entrée $I = \Gamma$ et de sortie $O = \Gamma \times \{\blacktriangleleft, \blacktriangledown, \blacktriangleright\}$. Le ruban est potentiellement infini mais, pour le calcul, il reste à chaque étape du calcul uniformément blanc sauf en un nombre fini de cases déjà visitées par la tête.*

3.2 Reconnaissance de langages

Le calcul de fonction modélise les procédures effectives de calcul. Le modèle des machines de Turing est cependant principalement utile pour obtenir des résultats d'impossibilité, d'incalculabilité (car sinon, à quoi bon définir un modèle qui capture toutes les fonctions calculables ?) Dans ce contexte, pour établir des résultats de borne inférieure et comparer des difficultés calculatoires, il est souvent plus pratique de se placer dans le cadre des prédicats calculables, de la reconnaissance de langages.

Remarque 3.6. *À tout langage $L \subseteq \Gamma^*$ est associée sa fonction caractéristique $\chi_L : \Gamma^* \to \{0, 1\}$ telle que $\chi_L(u) = 1$ si et seulement si $u \in L$. À toute fonction $f : \Sigma^* \rightharpoonup \Gamma^*$ est associé son graphe $G(f) = \{(u, f(u)) | u \in \mathrm{Dom}(f)\}$.*

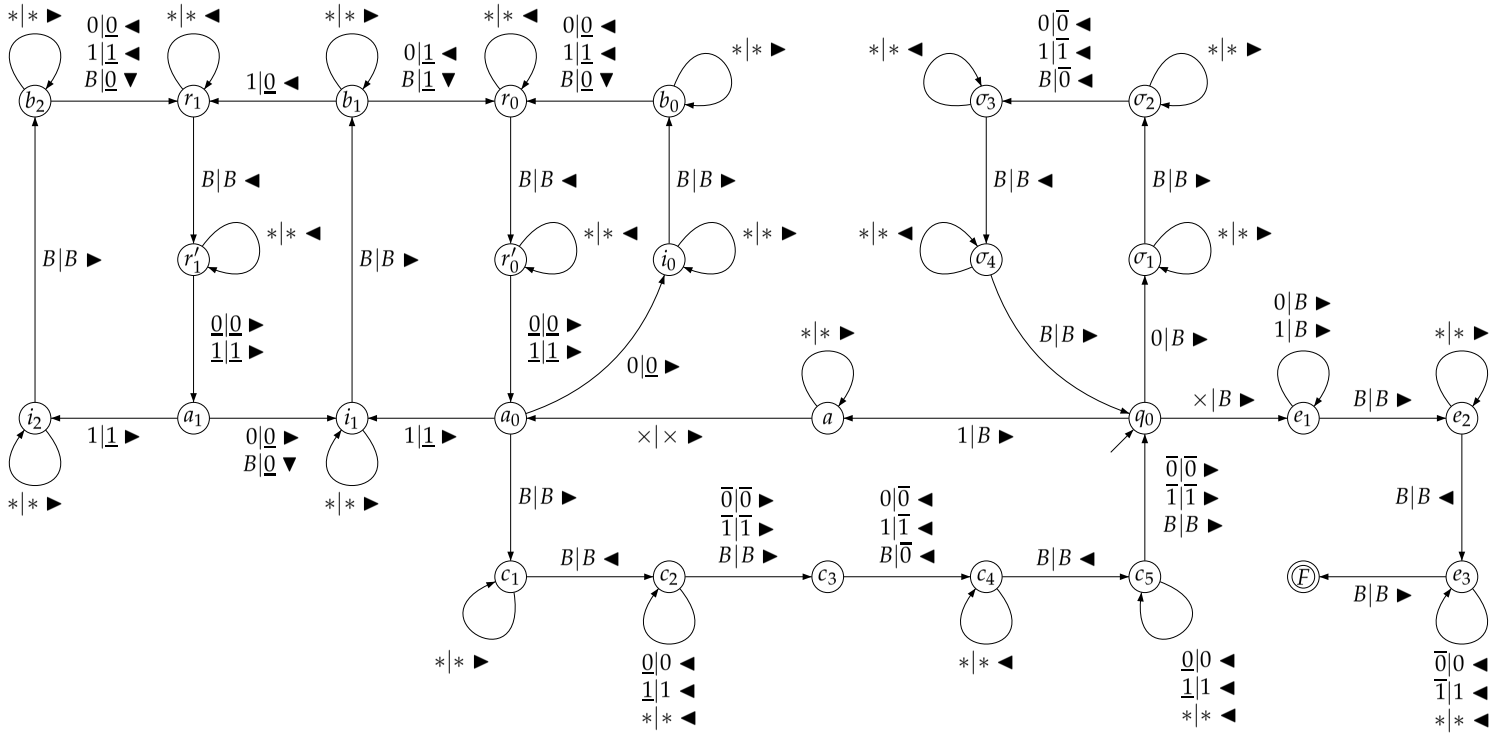


FIGURE 6 – *Multiplication d'entiers codés en binaire bit de poids faible en tête. Depuis l'état initial q_0, sur l'entrée* 011011×0111011*, cette machine de Turing s'arrête sur la sortie* 0010110011101*. Les transitions abrégées $*|*$ signifient $\alpha|\alpha$ pour tout symbole α pour lequel aucune transition n'est encore définie sur cet état.*

Une machine de Turing reconnaît un mot d'entrée u si le calcul associé est acceptant. La sortie est ignorée. Les mot rejetés, avec ou sans divergence, ne sont pas acceptés.

Définition 3.7. *Le langage $L(\mathcal{M}) \subseteq \Sigma^*$ reconnu par une machine de Turing $\mathcal{M}$ d'alphabet d'entrée Σ est l'ensemble des mots acceptés par la machine.*

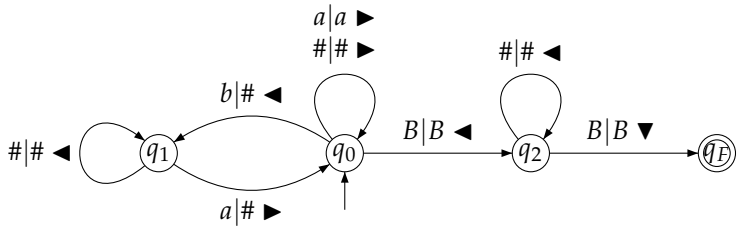


FIGURE 7 – *Machine de Turing totale reconnaissant les mots bien parenthésés sur l'alphabet $\{a, b\}$.*

À titre d'exemple, la machine de la figure 7 reconnaît les mots bien parenthésés sur l'alphabet $\{a, b\}$, c'est-à-dire les mots qui contiennent autant de a que de b et tels que pour tout préfixe, le nombre de a est supérieur ou égal au nombre de b.

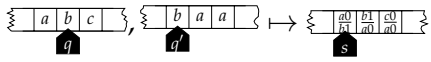
Définition 3.8. *Un langage est* récursivement énumérable *s'il est reconnu par une machine de Turing ;* co-récursivement énumérable *si son complémentaire est récursivement énumérable ;* récursif *s'il est reconnu par une machine de Turing totale.*

Lors du choix d'une machine pour reconnaître un langage qui est récursivement énumérable ou récursif, il est toujours possible de *normaliser* la machine choisie en demandant qu'elle possède en plus de l'état d'acceptation un état de rejet q_R de sorte qu'aucune transition ne soit définie pour ces deux états et que les seules transitions non définies soient celles-ci. C'est-à-dire que δ soit définie exactement sur $(Q \setminus \{q_F, q_R\}) \times \Gamma$.

Proposition 3.9. *La famille des langages récursifs est close par complémentaire.*

Démonstration. Soit L un langage récursif et $\mathcal{M}$ une machine de Turing totale normalisée qui le reconnaît. La machine de Turing $\mathcal{M}'$ obtenue à partir de $\mathcal{M}$ en gardant la même fonction de transition δ mais en prenant comme état d'acceptation l'état de rejet q_R est une machine de Turing totale qui reconnaît $\overline{L} = \Sigma^* \setminus L$. En effet, comme $\mathcal{M}$ est totale, pour toute entrée $u \in \Sigma^*$, le calcul associé s'arrête en q_R si et seulement si $u \notin L(\mathcal{M}) = L$. ■

La programmation concurrente est possible dans le monde des machines de Turing. Étant données deux machines $\mathcal{M}_1$ et $\mathcal{M}_2$ de même alphabet d'entrée Σ, on construit une machine $\mathcal{M}$ qui travaille sur l'alphabet $\Sigma \cup \{B\} \cup \Gamma_1 \times \Gamma_2 \times \{0,1\}^2$. Cette machine simule le fonctionnement concurrent de $\mathcal{M}_1$ et de $\mathcal{M}_2$ en effectuant alternativement des pas de calcul de chacune des deux machines. Pour simuler deux rubans dédiés, elle utilise les symboles de $\Gamma_1 \times \Gamma_2 \times \{0,1\}^2$: un symbole pour la piste de $\mathcal{M}_1$, un symbole pour la piste de $\mathcal{M}_2$ et deux bits pour indiquer si la tête d'une des deux machines pointe actuellement sur cette case :



Au début du calcul, $\mathcal{M}$ transforme le mot d'entrée u en le mot $(u_1, u_1, 1, 1)(u_2, u_2, 0, 0) \cdots (u_n, u_n, 0, 0)$ et passe dans l'état qui simule la paire de machines depuis leurs états initiaux respectifs. Pour simuler un pas de calcul, $\mathcal{M}$ se déplace dans la zone double du ruban jusqu'à trouver la tête de la machine à simuler et effectue une transition en ne modifiant que la portion du ruban qui concerne la machine simulée. Les états des machines $\mathcal{M}_1$ et $\mathcal{M}_2$ sont mémorisés dans l'état de $\mathcal{M}$. Si au cours du calcul $\mathcal{M}$ rencontre un symbole B, elle le transforme en $(B, B, 0, 0)$ avant de reprendre la simulation. Cette simulation concurrente d'une paire de machines permet d'établir aisément les deux propositions suivantes.

Proposition 3.10. *Un langage est récursif si et seulement s'il est récursivement énumérable et co-récursivement énumérable.*

Démonstration. Un langage récursif est reconnu par une machine de Turing, il est donc récursivement énumérable. D'après la proposition précédente, il en est de même pour son complémentaire, le langage est donc aussi co-récursivement énumérable. Réciproquement, en simulant de manière concurrente deux machines de Turing qui reconnaissent respectivement un langage et son complémentaire, on construit une machine de Turing totale qui reconnait le langage : elle s'arrête dès que l'une des deux machines simulées s'arrêtent et accepte uniquement si le mot est reconnu par la première machine. ■

Proposition 3.11. *La famille des langages récursivement énumérables est close par union et intersection.*

Démonstration. Étant donnés deux languages récursivement énumérables, on construit une machine qui reconnaît leur intersection, respectivement leur union, en simulant de manière concurrente deux machines qui reconnaissent ces langages et en acceptant lorsque les deux machines simulées ont accepté, respectivement dès que l'une des deux machines accepte. ■

Corollaire 3.12. *La famille des langages récursifs est close par union et intersection.*

Par un simple argument de cardinalité, on montre qu'il existe des langages non récursifs.

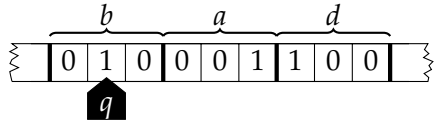
Proposition 3.13. *Il existe des langages non récursifs.*

Démonstration. L'ensemble des machines de Turing est dénombrable alors que l'ensemble des langages ne l'est pas. ■

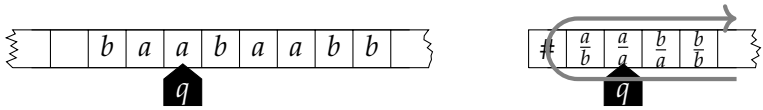
3.3 Variations sur les mémoires

La puissance de calcul des machines de Turing provient de leur capacité à stocker, marquer, accéder à une information de taille arbitraire. En remplaçant le ruban biinfini par d'autres types de mémoires, la famille des fonctions calculables reste la même, éventuellement à un recodage calculable près sur les entrées et sorties. Voici quelques variantes dont le lecteur pourra se convaincre facilement qu'ils sont simulables par machine de Turing et, avec les arguments donnés dans le texte, qu'elles capturent toutes les fonctions Turing-calculables.

Ruban binaire [$\delta : Q \times \{0,1\} \rightharpoonup Q \times \{0,1\} \times \{\blacktriangleleft, \blacktriangledown, \blacktriangleright\}$] L'alphabet de travail peut, après recodage des entrées, être troqué pour un ruban binaire. Il suffit à la machine de lire ou d'écrire plusieurs bits successifs pour se ramener à un alphabet plus grand.

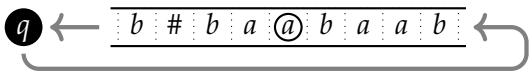


Ruban monoinfini Une machine à ruban infini uniquement à droite et pour laquelle le calcul s'arrête si la tête sort du ruban. Il suffit à la machine de replier le ruban biinfini d'une machine classique pour le simuler : la case i reçoit les symboles des cases $(-i+1, i)$, la case 0 reçoit un symbole spécial de bord de ruban, la machine utilise son état pour mémoriser le demi-ruban sur lequel elle travaille.



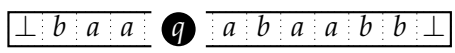
Rubans multiples [$\delta : Q \times \Gamma^k \rightharpoonup Q \times (\Gamma \times \{\blacktriangleleft, \blacktriangledown, \blacktriangleright\})^k$] Le pilotage de plusieurs rubans permet de simplifier le codage de certains algorithmes et de séparer les zones de lecture d'entrée et d'écriture de sortie. La simulation de plusieurs rubans avec un seul a déjà été décrite lors de la simulation concurrente de deux machines.

File [$\delta : Q \times (\Gamma \cup \{\bot\}) \rightharpoonup Q \times \Gamma^*$] Le ruban peut être remplacé par une file. À chaque transition, l'automate reçoit le premier élément de la file ou un symbole de file vide et il fournit un mot à enfiler. La portion non blanche d'un ruban peut être représentée par une file sur l'alphabet $\{\#\} \cup \Gamma \times \{0, 1\}$ où le symbole $\# \notin \Gamma$ indique le bord du ruban et le bit supplémentaire permet d'identifier la case sur laquelle pointe la tête de lecture/écriture. L'automate fait tourner la configuration, défilant et enfilant les éléments jusqu'à accéder à la case pointée pour y exécuter une transition. La portion considérée grandit par ajout de symboles blancs dans la file.

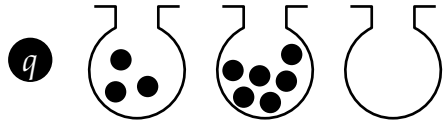


Ruban non modifiable Un ruban non modifiable est un ruban où chaque case peut être modifiée au plus une fois. Sur un tel ruban, une file peut être simulée en utilisant deux cases par élément dans la file. Deux cases blanches indiquent la fin de la file. La première case blanche et l'autre non indique une case active de la file. Lorsqu'un symbole est défilé, sa première case est modifiée. Lorsqu'un symbole est enfilé, une case est laissée blanche et le symbole à enfiler est inscrit dans la case suivante. L'automate installe le mot d'entrée dans la file en début de calcul puis simule le comportement d'un automate à file.

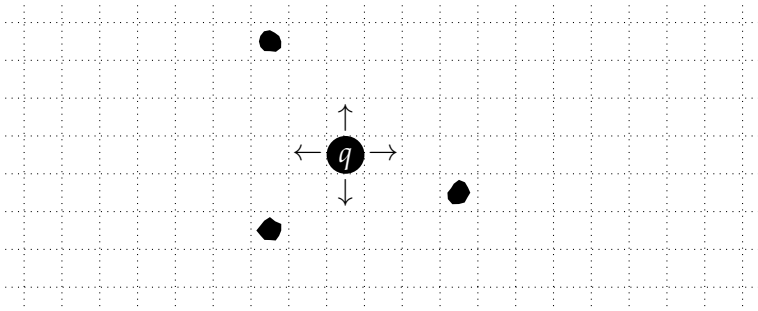
Piles multiples [$\delta : Q \times (\Gamma \cup \{\bot\})^k \rightharpoonup Q \times (\Gamma^*)^k$] Le ruban peut être remplacé par une ou plusieurs piles. Dans ce scénario, le mot d'entrée peut être placé initialement sur une pile ou accédé séquentiellement comme un élément supplémentaire de I. Deux piles suffisent pour simuler un ruban de machine de Turing. En effet, une configuration uqv où $u = u_n \cdots u_1$ et $v = v_1 \cdots v_m$ peut être codée en plaçant u sur une première pile avec u_1 en tête de pile et v sur une deuxième pile avec v_1 en tête de pile. La machine a accès à v_1 et q pour calculer la transition et peut déplacer des symboles d'une pile à l'autre pour simuler un déplacement de la tête sur le ruban. Lorsqu'elle atteint le fond de pile, la machine simule la lecture d'un symbole blanc.



Compteurs multiples [$\delta : Q \times \{0,+\}^k \rightharpoonup Q \times \{-1,0,+1\}^k$] Le ruban peut être remplacé par un ou plusieurs compteurs. Chaque compteur stocke un entier, l'automate teste si le compteur est à zéro ou non et agit sur le compteur en le décrémentant ou en l'incrémentant. Autrement dit, il s'agit de la jarre de galets de la bergère. Avec un compteur de travail, initialement à 0, il est possible de transformer la valeur n stockée dans un compteur : multiplication par une constante, division par une constante avec mémorisation du reste de la division par l'automate. Cela suffit à simuler une pile : la pile sur un alphabet de taille k est vue comme un entier écrit en base $k+1$, les opérations sur la pile se ramènent à des multiplications et divisions par $k+1$. Ainsi avec trois compteurs, la machine simule deux piles donc un ruban Turing. M. Minsky [22] décrit un codage plus astucieux qui permet, au prix d'un codage exponentiel de l'entrée, de simuler une machine de Turing avec seulement deux compteurs.



Galets dans le plan [$\delta : Q \times \{0,1\}^k \rightharpoonup Q \times \{0,1\}^k \times \boxplus$] avec $\boxplus = \{(-1,0),(1,0),(0,0),(0,-1),(0,1)\}$. Le ruban peut être remplacé par une grille $\mathbb{Z}^2$ sur laquelle la machine se déplace en ayant la possibilité de poser et de ramasser des galets[8] pris dans une collection finie. La distance entre deux galets permet de coder un entier dont il est facile de tester s'il vaut zéro et de l'incrémenter ou de le décrémenter. Avec trois galets on obtient un automate à deux compteurs et donc une machine qui calcule toutes les fonctions Turing-calculables.



4 Calculer autrement

Comme on vient de le voir, la notion de fonction calculable reste robuste à des variations. Pour autant, il s'agit à chaque fois d'un automate

fini contrôlant une mémoire non bornée. D'autres formes de calcul sont possible, par exemple le modèle du λ-calcul proposé par Church dès 1936. Une découverte majeure de la calculabilité depuis les années 30 est que la notion est universelle. Tous les modèles de calcul que l'on conçoit comme des méthodes effectives de calcul et qui sont suffisamment puissants définissent une même et unique notion unifiée de fonction calculable. Cette thèse (ce n'est pas un théorème car la notion de méthode effective n'est pas formalisable) porte le nom de thèse de Church-Turing[9].

Thèse 4.1 (A. Church, A. Turing, S. Kleene, E. Post *et al.*)**.** *Toute fonction calculable par une méthode effective est Turing-calculable.*

Différentes formalisations partielles ont été proposées afin d'appuyer cette thèse. En particulier, des démonstrations sont possibles dans le cadre de la physique théorique.

Théorème 4.2 (R. Gandy [12])**.** *Toute fonction discrète calculable par un dispositif mécanique déterministe, régit par les règles de la physique classique et qui satisfait les hypothèses suivantes, est Turing-calculable :*

— *homogénéité de l'espace ;*
— *homogénéité du temps ;*
— *densité bornée d'information dans l'espace ;*
— *vitesse bornée de propagation de l'information à travers l'espace ;*
— *quiescence initiale de l'espace de calcul sauf dans une zone bornée.*

Une version de ce théorème transposée dans le monde de la physique quantique a été étudiée par P. Arrighi et G. Dowek [2], qui montrent que sous de bonnes hypothèses le théorème reste vrai.

Un *modèle de calcul* décrit une famille dénombrable d'objets qui calculent et la manière de les utiliser pour mener à bien un calcul : comment encoder l'entrée du calcul, comment réaliser des pas de calcul, comment décider de l'arrêt du calcul et comment décoder la sortie. Pour comparer des modèles de calcul, il faut expliciter la manière de transposer une fonction d'un modèle à un autre. En effet, certains modèles travaillent sur des mots sur un alphabet fixé, d'autres sur des entiers, d'autres encore sur des arbres binaires, etc. Cette transposition passe par une fonction de codage.

Une *fonction de codage acceptable* transforme les objets de manière calculable (par un modèle de calcul qui manipule les deux types d'objets) de sorte que :

— la transformation soit injective ;
— l'image de l'ensemble des objets forme un langage récursif ;

— les opérations élémentaires (accéder à une lettre pour un mot, incrémenter ou décrémenter un entier, ...) sur l'objet codé puissent être mises en œuvre de manière calculable sur l'objet d'arrivé.

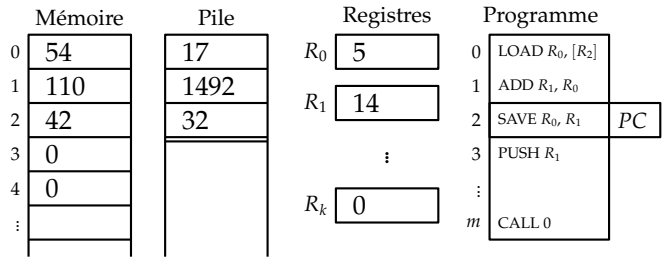
Tous les codages acceptables étant compatibles et d'une certaine manière équivalents, la notation $\langle\cdot\rangle$ est utilisée pour fixer un codage acceptable arbitraire lorsque nécessaire.

Définition 4.3. *Deux modèles de calculs sont* équivalents *s'ils se simulent entre eux, c'est-à-dire s'ils calculent les mêmes fonctions à un codage acceptable près.*

Définition 4.4. *Un modèle de calcul est* Turing-complet *s'il est équivalent au modèle des machines de Turing.*

En plus des modèles d'automates couplés à des mémoires bornées, voici quelques exemples de modèles de calcul Turing-complet.

Modèle RAM Les processeurs de nos ordinateurs sont limités par l'espace disponible et travaillent avec des registres et des mots limités à un nombre fixé de bits. Le modèle RAM propose de programmer des processeurs qui travaillent sur des registres capables de stocker des entiers de taille arbitraire avec une mémoire de capacité non bornée. Dans une version complète, on y trouve plusieurs zones mémoire : un programme écrit sur un jeu d'instruction expressif (architecture de Harvard avec mémoire séparée pour le programme), un jeu fini de registres $PC, R_0, \ldots, R_k$, une pile pour gérer les appels récursifs, une mémoire adressable potentiellement infinie. Ce modèle est très redondant, sans mémoire ni pile, avec seulement 3 registres et un jeu d'instructions limité, on retrouve les automates à compteurs qui sont déjà Turing-complets.



Programmation impérative De même que pour les processeurs, tout langage de programmation peut être transformé en un modèle de calcul Turing-complet en autorisant la manipulation de ressources non bornées. La figure 8 propose la syntaxe d'un langage impératif IMP dont la sémantique est laissée au lecteur. Ce langage qui manipule des entiers non bornés

Variables	$\ni$	X, Y		
Expressions	$\ni$	E, F	$::=$	X
			$\mid$	Constante entière
			$\mid$	$E \odot F$ avec $\odot \in \{+, -, *, \texttt{DIV}, \texttt{MOD}\}$
			$\mid$	$E \odot F$ avec $\odot \in \{=, <, >, \texttt{AND}, \texttt{OR}\}$
			$\mid$	$\odot E$ avec $\odot \in \{\texttt{NOT}, -\}$
Instructions	$\ni$	S, T	$::=$	$X = E$
			$\mid$	S ; T
			$\mid$	IF E THEN S ELSE T
			$\mid$	WHILE E DO S
Programme	$\ni$	P	$::=$	READ X; S; WRITE Y

FIGURE 8 – *Syntaxe du langage imaginaire IMP*

se compile sans difficulté vers le modèle RAM ci-dessus. Le lecteur intéressé par une approche de la calculabilité plus proche de la programmation est invité à consulter l'ouvrage de N. Jones [15].

λ-calcul et combinateurs A. Church [4] propose comme méthode effective de calcul la normalisation des termes du λ-calcul. Un λ-terme est défini par induction par la formule $\Lambda ::= x | \lambda x.\Lambda | (\Lambda\ \Lambda')$ où x désigne un nom de variable parmi un ensemble dénombrable. La normalisation se définit à partir de la notion d'α-conversion et de substitution. Nous préférons ici définir le calcul de combinateurs SK qui lui est équivalent : tout λ-terme peut être codé par un SK-terme et réciproquement. Un SK-terme est défini par induction comme un arbre binaire étiqueté par les symboles S et K sur les feuilles, *i.e.* $\alpha, \beta ::= S | K | (\alpha\ \beta)$. Pour normaliser un SK-terme, deux règles de réécriture sont appliquées tant que possible sur les sous-arbres :

$$\begin{aligned} ((K\alpha)\beta) &\Rightarrow \alpha \\ (((S\alpha)\beta)\gamma) &\Rightarrow ((\alpha\gamma)(\beta\gamma)) \end{aligned}$$

Dans ce modèle de calcul, programmes et données sont des termes et un calcul est une normalisation.

Fonctions récursives à la J. Herbrand, K. Gödel, S. Kleene [17] Le modèle de calcul des fonctions récursives décrit l'ensemble des fonctions partielles calculables $f : \mathbb{N}^k \rightharpoonup \mathbb{N}$ à arguments entiers et valeur dans les entiers. Ces fonctions sont définies comme le plus petit ensemble contenant la fonction successeur, les constantes et les projections et clos par trois opérateurs : la composition, le schéma de récursivité primitive et le schéma de minimisation. La figure 9 décrit formellement ces fonctions et schémas. A. Turing

$$
\begin{aligned}
S(x) &= x+1 \\
\mathrm{C}_n^k(x_1,\ldots,x_k) &= n && \forall k,n \in \mathbb{N} \\
\pi_n^k(x_1,\ldots,x_k) &= x_n && \forall k,n \in \mathbb{N}
\end{aligned}
$$

$$
\begin{aligned}
(h \circ (g_1,g_2,\ldots,g_m))\,(x_1,\ldots,x_k) = h(&g_1(x_1,\ldots,x_k),\\
&g_2(x_1,\ldots,x_k),\\
&\ldots,\\
&g_m(x_1,\ldots,x_k)) && \forall h,g_1,\ldots,g_m
\end{aligned}
$$

$$
\begin{aligned}
\rho_g^h(0,x_1,\ldots,x_k) &= g(x_1,\ldots,x_k) \\
\rho_g^h(S(x_0),x_1,\ldots,x_k) &= h(x_0,\\
&\qquad \rho_g^h(x_0,x_1,\ldots,x_k),\\
&\qquad x_1,\ldots,x_k) && \forall g,h
\end{aligned}
$$

$$
\mu f(x_1,\ldots,x_k) = \min\{x_0 | f(x_0,x_1,\ldots,x_k) = 0\} \qquad \forall f
$$

FIGURE 9 – *Fonctions récursives : fonctions élémentaires S, C_n^k, π_n^k, schémas de composition $\circ$, de récursivité primitive ρ_g^h et de minimisation μf.*

démontre dans [37] l'équivalence entre les machines de Turing, le λ-calcul et les fonctions récursives.

Algorithmes de A. Markov Le modèle des algorithmes de A. Markov [19] est un modèle de réécriture sur les mots. Un algorithme de Markov est la donnée d'un alphabet d'entrée Σ et d'une suite finie de règles de réécriture $u_i \to v_i$, où $u_i, v_i \in \Gamma^*$ sont des mots sur un alphabet $\Gamma \supseteq \Sigma$. Certaines des règles sont identifiées comme terminales $u_i \to\cdot\, v_i$ (noter le point). À partir d'un mot d'entrée u, on itère le procédé suivant. À chaque pas de calcul, on identifie le premier des u_i qui est un facteur du mot courant et on substitue son occurrence la plus à gauche par v_i. Lorsque la règle $u_i \to v_i$ appliquée est terminale ou lorsqu'aucune règle ne s'applique, le calcul s'arrête et le mot courant est la sortie du calcul. Ce modèle de calcul simule aisément une machine de Turing en utilisant les descriptions instantanées pour coder les configurations de la machine.

Systèmes canoniques d'E. Post (Tag systems) Le modèle des *Tag systems* d'E. Post [29] est un autre modèle de réécriture sur les mots, très utilisé pour construire de petits systèmes Turing-complets. Un système k-canonique est un triplet (Σ, k, φ) où Σ est un alphabet fini, k est un entier positif et

$\varphi : \Sigma \rightharpoonup \Sigma^*$ est une fonction partielle de transition. Une configuration du modèle est un mot sur Σ. Un pas de calcul consiste à effacer les k premières lettres du mot et à y concaténer l'image de la première lettre effacée par φ, *i.e.* $u_1 \cdots u_k v \vdash v\varphi(u_1)$. Le calcul s'arrête lorsqu'il reste strictement moins de k lettres ou lorsque φ n'est pas défini sur la première lettre du mot. Une valeur de $k = 2$ suffit pour simuler une machine de Turing [5].

Automates cellulaires Les automates cellulaires sont des systèmes dynamiques discrets parfois considérés comme des modèles du parallélisme massif. Au lieu de coupler un automate à une mémoire, un automate cellulaire est une collection régulière, sur une grille, d'une infinité potentielle de copies d'un même automate fini. Le système est synchrone et uniforme. À chaque pas de calcul, chacun des automates met à jour son état de manière déterministe grâce à l'observation de son propre état et des états des automates voisins. Ainsi, en dimension 1, pour une ligne d'automates observant leurs deux plus proches voisins, la fonction de transition est de la forme $\delta : Q^3 \to Q$. Une configuration du système est un coloriage de la grille par les états, *i.e.* $c \in Q^{\mathbb{Z}}$. La fonction globale de transition $F : Q^{\mathbb{Z}} \to Q^{\mathbb{Z}}$ réalise un pas de calcul, *i.e.* $F(c)(i) = f(c(i-1), c(i), c(i+1))$ pour tout $c \in Q^{\mathbb{Z}}$ et $i \in \mathbb{Z}$. Pour calculer avec un tel système dynamique, il est essentiel de considérer avec précaution la manière de coder l'entrée, de détecter l'arrêt et de décoder la sortie. Le lecteur intéressé pourra consulter l'état de l'art de l'auteur [25]. En simulant une variante des systèmes canoniques d'E. Post, M. Cook [6] démontre l'universalité pour le calcul d'un automate cellulaire de dimension 1 à seulement 2 états, la « règle 110 ». Le chapitre **??** explore les liens entre systèmes dynamiques et calcul.

5 Machines programmables

Dans le modèle de calcul des machines de Turing, chaque méthode effective, chaque fonction calculable, est réalisée par un automate fini couplé à une mémoire. A priori, cet automate doit être modifié pour chaque calcul. Cependant, et c'est là une deuxième contribution significative de l'article d'A. Turing [36], ce modèle de calcul mécanique est si simple qu'il permet de réaliser un fait essentiel : les programmes sont des données comme les autres. Il existe une machine de Turing universelle U qui, à partir d'une description d'une machine de Turing $\mathcal{M}$ et d'une entrée u pour cette machine, simule le calcul de $\mathcal{M}$ sur u. Autrement dit, il suffit de construire

une machine de Turing universelle, dont l'automate est fixé une fois pour toutes, pour être capable de calculer toute fonction Turing-calculable.

Théorème 5.1. *Il existe une machine de Turing universelle U qui simule toutes les machines de Turing. Pour toute machine de Turing $\mathcal{M}$ sur l'alphabet Σ et toute entrée $u \in \Sigma^*$, la machine U s'arrête sur l'entrée $\langle \mathcal{M}, u \rangle$ si et seulement si $\mathcal{M}$ s'arrête sur l'entrée u. Elle accepte si et seulement si $\mathcal{M}$ accepte et dans ce cas*

$$f_U\left(\langle \mathcal{M}, u \rangle\right) = \langle f_{\mathcal{M}}(u) \rangle \quad .$$

L'utilisation de codages acceptables dans la définition ci-dessus est nécessaire : la machine U travaille sur un alphabet fixé et simule des machines sur des alphabets de taille arbitraire. Comment fonctionne une telle machine U ?

Nous décrivons ci-dessous le principe de fonctionnement d'une telle machine sans chercher à optimiser la taille de son alphabet ou la complexité de U (en nombre de transitions, vitesse de calcul, etc). À titre d'exemple, nous illustrons son calcul sur l'entrée $\langle \mathcal{M}, bbab \rangle$ où $\mathcal{M}$ est la machine de Turing de la figure 10.

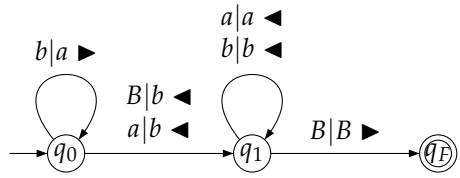
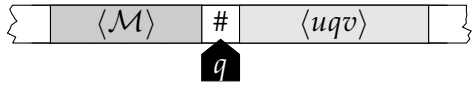


FIGURE 10 – *Machine de Turing incrémentant un entier.*

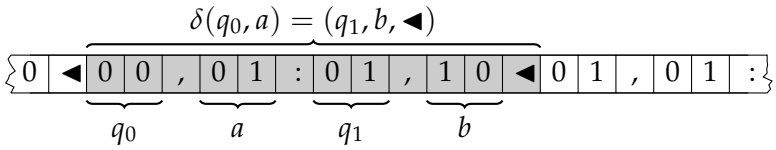
Le ruban de U est organisé en deux parties : une description $\langle \mathcal{M} \rangle$ de la machine simulée séparée par un symbole # d'une description de la configuration courante $\langle uqv \rangle$.



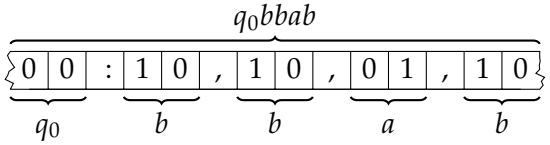
Les états et les symboles sont codés sur un nombre suffisant de bits avec la convention $\langle B \rangle = 0 \cdots 0$, $\langle q_0 \rangle = 0 \cdots 0$, $\langle q_F \rangle = 1 \cdots 1$. Dans notre exemple les états q_0, q_1 et q_F sont codés respectivement par 00, 01 et 11 ; les symboles B, a et b sont codés respectivement par 00, 01 et 10.

L'élément essentiel de $\mathcal{M}$ pour la simulation est la fonction de transition : les quintuplets (q, a, q', b, Δ) pour lesquels $\delta(q, a) = (q', b, \Delta)$. Le codage de la machine est la suite des codages des quintuplets et chaque

quintuplet est codé par ses composants en interposant des séparateurs entre deux entiers pour les délimiter.



Le codage de la configuration $\langle uqv \rangle$ repose sur le même principe : chaque élément de $Q \cup \Gamma$ est codé en binaire avec un séparateur entre chaque élément. Le séparateur entre q et v_1 est différencié pour identifier la position de la tête.



La simulation d'un pas de calcul avec ce codage est très simple. La machine U identifie la paire (q, a) courante de $\mathcal{M}$ dans la configuration et la cherche dans la table de transition. Une fois la transition identifiée, la configuration est mise à jour et la tête déplacée. Si le bord de la description instantanée est atteint, un symbole blanc est inséré. Lorsqu'aucune transition n'est plus possible, la machine U nettoie la configuration pour ne garder que la sortie codée et elle accepte si et seulement si l'état final de $\mathcal{M}$ était l'état d'acceptation.

Au début de la simulation, sur l'entrée $\langle \mathcal{M}, u \rangle$, la machine U insère $\langle q_0 \rangle$ dans la configuration et la simulation peut commencer.

Le lecteur intéressé par la construction de petites machines universelles capables de simuler efficacement toutes les machines de Turing est invité à consulter l'article de D. Woods et T. Neary [40].

6 Indécidabilité et limites du calcul

Avec la machine universelle, et donc avec la capacité de décrire des machines de Turing qui manipulent des descriptions d'autres machines de Turing et simulent leur calcul jusqu'à ce qu'elles s'arrêtent, nous disposons de tous les éléments pour développer la notion de problème indécidable. Un problème de décision n'est ni plus ni moins qu'un langage débarassé des détails techniques liés aux codages acceptables.

Définition 6.1. *Un* problème de décision $\mathcal{P}$ *est un prédicat sur un ensemble récursif d'entrées E, les instances du problème. Le langage associé au problème est le langage des codages des instances positives, i.e.* $L_{\mathcal{P}} = \{\langle x \rangle | x \in E \wedge \mathcal{P}(x)\}$.

Le plus célèbre problème de décision est sans conteste le problème de l'arrêt des machines de Turing :

PROBLÈME DE L'ARRÊT

entrée : *une machine de Turing* $\mathcal{M}$ *et un mot* u
question : *est-ce que* $\mathcal{M}$ *s'arrête sur l'entrée* u *?*

Le langage associé à ce problème est $K = \{\langle \mathcal{M}, u \rangle | \mathcal{M}$ s'arrête sur $u\}$.

Un problème de décision est décidable s'il existe un programme capable de résoudre ce problème. Dans le cas contraire, le problème est indécidable.

Définition 6.2. *Un problème de décision est* décidable *si le langage associé est récursif,* indécidable *sinon.*

6.1 Un premier problème indécidable

Le problème de l'arrêt est-il décidable ? La construction de la machine universelle nous permet d'affirmer que K est récursivement énumérable. A. Turing [36] établit son indécidabilité en utilisant un argument diagonal.

Théorème 6.3. *Le problème de l'arrêt est indécidable.*

Démonstration. Supposons que le problème soit décidable et soit $\mathcal{H}$ une machine de Turing totale qui reconnaît K. On construit une machine de Turing Δ à partir de $\mathcal{H}$. Sur l'entrée $\langle \mathcal{M} \rangle$, la machine Δ ainsi définie exécute les transitions de $\mathcal{H}$ sur l'entrée $\langle \mathcal{M}, \mathcal{M} \rangle$. Si $\mathcal{H}$ accepte alors Δ diverge. Si $\mathcal{H}$ rejette alors Δ accepte. La machine Δ possède elle-même un codage $\langle \Delta \rangle$. Que se passe-t-il si on invoque Δ sur l'entrée $\langle \Delta \rangle$?

- si $\mathcal{H}(\langle \Delta, \Delta \rangle)$ accepte, c'est-à-dire si Δ s'arrête sur $\langle \Delta \rangle$, alors Δ diverge… donc Δ diverge sur $\langle \Delta \rangle$, contradiction ;
- si $\mathcal{H}(\langle \Delta, \Delta \rangle)$ rejette, c'est-à-dire si Δ diverge sur $\langle \Delta \rangle$, alors Δ s'arrête… donc Δ s'arrête sur $\langle \Delta \rangle$, contradiction.

L'existence d'une telle machine $\mathcal{H}$ est impossible. ■

Corollaire 6.4. *Le langage K n'est pas co-récursivement énumérable.*

6.2 Réductions many-one

En munissant l'ensemble des langages de relations de préordre bien choisies, compatibles avec le calcul, les réductions, on obtient un outil très pratique pour dériver des résultats d'indécidabilité à partir de problèmes déjà connus pour être indécidables. Cette méthode évite de répéter les arguments de diagonalisation. La réduction la plus courante est la réduction *many-one* (cette désignation indique qu'on n'impose pas l'injectivité de la fonction de réduction).

Définition 6.5. *Soient $A \subseteq \Sigma^*$ et $B \subseteq \Gamma^*$ deux langages. Une* réduction *(many-one) de A à B est une fonction totale calculable $f : \Sigma^* \to \Gamma^*$ telle que*

$$u \in A \Leftrightarrow f(u) \in B \qquad \forall u \in \Sigma^* \quad .$$

Définition 6.6. *Le langage A se réduit au langage B, noté $A \leqslant_m B$ s'il existe une réduction de A à B.*

Proposition 6.7. *La relation $\leqslant_m$ est une relation de préordre.*

Démonstration. Soient A, B et C trois langages. La relation $\leqslant_m$ est réflexive car la fonction identité est une réduction de A à lui-même. Elle est transitive car si f est une réduction de A à B et g une réduction de B à C alors $g \circ f$ est une réduction de A à C : les fonctions totales calculables sont stables par composition. ■

Proposition 6.8. *Si A se réduit à B alors $\overline{A}$ se réduit à $\overline{B}$.*

Démonstration. Une réduction de A à B est aussi une réduction pour leurs complémentaires, de $\overline{A}$ à $\overline{B}$, il suffit de considérer la contraposée de la définition. ■

Proposition 6.9. *Si A se réduit à B et si B est récursivement énumérable alors A l'est aussi.*

Démonstration. Soit A et B deux langage et f une réduction de A à B. Si B est reconnu par une machine de Turing $\mathcal{B}$ alors A est reconnu par la machine $\mathcal{A}$ qui, sur l'entrée u, calcule $f(u)$ puis exécute $\mathcal{B}$ dessus. ■

Corollaire 6.10. *Si $\mathcal{P}$ est indécidable et $L_{\mathcal{P}} \leqslant_m L_{\mathcal{P}'}$ alors $\mathcal{P}'$ est indécidable.*

À titre d'exemple, montrons que le problème de l'arrêt sur l'entrée vide est indécidable en utilisant une réduction.

PROBLÈME DE L'ARRÊT SUR L'ENTRÉE VIDE

entrée : *une machine de Turing* $\mathcal{M}$
question : *est-ce que* $\mathcal{M}$ *s'arrête sur l'entrée vide* ε *?*

Le langage associé à ce problème est $K_0 = \{\langle \mathcal{M} \rangle | \mathcal{M} \text{ s'arrête sur } \varepsilon\}$.

Pour montrer que ce problème est indécidable, il suffit d'exhiber une réduction f de K à K_0 qui à une entrée $\langle \mathcal{M}, u \rangle$ associe $\langle \mathcal{N} \rangle$ le codage d'une machine de Turing qui s'arrête sur l'entrée vide si et seulement si $\mathcal{M}$ s'arrête sur l'entrée u. La fonction f transforme $\mathcal{M}$ en une machine $\mathcal{N}$ qui commence par inscrire u sur son ruban avant d'invoquer $\mathcal{M}$. Cette transformation est bien calculable à partir de $\mathcal{M}$ et de u.

Une machine de Turing est un objet syntaxique qui décrit la manière d'organiser un calcul. Le langage reconnu par la machine est de nature sémantique, c'est le résultat du calcul. Le théorème de H. Rice [32] montre que tout problème qui voudrait caractériser une propriété purement sémantique des machines est indécidable ou trivial.

Définition 6.11. *Une propriété* $\mathfrak{P}$ *des langages est* non triviale *s'il existe deux machines de Turing* $\mathcal{M}$ *et* $\mathcal{M}'$ *telles que* $L(\mathcal{M})$ *vérifie* $\mathfrak{P}$ *et* $L(\mathcal{M}')$ *ne vérifie pas* $\mathfrak{P}$*.*

Théorème 6.12. *Soit* $\mathfrak{P}$ *une propriété des langages non triviale. Le problème de décision ci-dessous est indécidable.*

PROBLÈME D'APPARTENANCE À $\mathfrak{P}$

entrée : une machine de Turing $\mathcal{M}$
question : est-ce que $L(\mathcal{M})$ vérifie $\mathfrak{P}$?

Démonstration. Soit $\mathfrak{P}$ une propriété des langages non triviale. Soit $\mathcal{Z}$ une machine de Turing qui diverge sur toute entrée. Sans perte de généralité, supposons que $L(\mathcal{Z})$ ne vérifie pas $\mathfrak{P}$ et soit $\mathcal{M}$ une machine telle que $L(\mathcal{M})$ vérifie $\mathfrak{P}$. Construisons une réduction f de K_0 à $L_{\mathfrak{P}}$. Sur l'entrée $\langle \mathcal{N} \rangle$, la fonction f construit une machine de Turing $\mathcal{N}'$ qui sur l'entrée u commence par exécuter $\mathcal{N}$ sur l'entrée vide. Lorsque le calcul de $\mathcal{N}$ s'arrête, la machine $\mathcal{N}'$ exécute le calcul de $\mathcal{M}$ sur l'entrée u. De sorte que, d'une part, si $\mathcal{N}$ ne s'arrête pas sur l'entrée vide alors $L(\mathcal{N}') = L(\mathcal{Z})$ et $\langle \mathcal{N}' \rangle \notin L_{\mathfrak{P}}$. Et d'autre part, si $\mathcal{N}$ s'arrête sur l'entrée vide alors $(L(\mathcal{N}') = L(\mathcal{M})$ et $\langle \mathcal{N}' \rangle \in L_{\mathfrak{P}}$. La fonction f est bien totale et calculable. ■

7 Pour aller plus loin : vers l'infini et au-delà !

7.1 Au-delà du récursif

Cette excursion dans le monde du calcul nous a menés devant les portes de l'indécidable, aux limites du calculable. C'est ici que la théorie de la calculabilité commence réellement, lorsque la question se pose de comparer les langages non récursifs. Mais comment comparer ces langages hors d'atteinte du modèle des machines de Turing ? L'ajout d'un oracle à la machine permet de franchir le seuil.

Définition 7.1. *Une machine de Turing avec oracle $L \subseteq \Gamma^*$ est une machine de Turing munie d'un ruban supplémentaire, le ruban d'oracle, accessible en lecture/écriture et de trois états particuliers $q_?$, q_o, q_n. À tout moment pendant le calcul, si la machine entre dans l'état $q_?$, le mot u situé sur le ruban d'oracle entre la position de la tête et le premier symbole B est testé instantanément : si $u \in L$ alors la machine passe dans l'état q_o, sinon elle passe dans l'état q_n.*

Les machines avec oracle récursif ne calculent rien de plus que leurs homologues classiques. Par contre, en utilisant un oracle non récursif L, on obtient une nouvelle famille de langages L-récursifs. Sur cette nouvelle famille de langages, le problème de l'arrêt est à nouveau indécidable : on construit un nouveau langage K^L… et on peut recommencer l'expérience. Une relation de préordre permet de comparer les langages dans ce cadre, la réduction Turing.

Définition 7.2. *Un langage $A \subseteq \Sigma^*$ se* Turing-réduit *à un langage $B \subseteq \Gamma^*$, noté $A \leqslant_T B$ si A est B-récursif, c'est-à-dire si A est reconnu par une machine de Turing totale avec oracle B.*

L'étude des degrés Turing, les classes d'équivalence de la relation $\leqslant_T$, permet de commencer l'exploration de l'indécidable. Le lecteur alléché est incité à consulter P. Odifreddi [24].

7.2 Calculer sur les réels

Nous avons ouvert ce chapitre avec des opérations arithmétique sur des nombres, addition, multiplication. Mais peut-on donner un sens à la calculabilité sur les nombres réels ? Cette question est déjà présente dans les travaux d'A. Turing. L'analyse calculable [39] s'intéresse à la question des réels calculable et des fonctions partielles $f : \mathbb{R} \rightharpoonup \mathbb{R}$ calculables.

Pour décrire un nombre réel de sorte à le rendre manipulable par une machine, on se ramène à une suite d'objets finis qui convergent vers ce

nombre. Cette suite d'objets n'est pas unique, d'ailleurs la comparaison de réels calculables n'est pas calculable.

Définition 7.3. *Un nombre réel $x \in \mathbb{R}$ est* défini *par une suite de nombres rationnels (q_i) si $|x - q_n| < 2^{-n}$ pour tout entier positif n.*

Définition 7.4. *Un nombre réel $x \in \mathbb{R}$ est calculable s'il existe une fonction Turing-calculable qui à tout entier positif i associe un rationnel $q_i \in \mathbb{Q}$ de sorte que la suite (q_i) définisse x.*

Une machine de Turing de type 2 est une machine de Turing munie, en plus de ses rubans de travail classiques, d'un ou plusieurs rubans d'entrée en lecture seule et d'un ruban de sortie en écriture seule qui accueillent des mots infinis. Ce type de machine calcule sans s'arrêter. Chaque ruban d'entrée reçoit un nombre réel calculable sous la forme de codages d'une suite de rationnels $\langle q_i \rangle$ qui définit un nombre réel x. Ces rationnels sont séparés par des symboles blancs. Sur les rubans d'entrée, la tête de lecture/écriture ne modifie pas le contenu du ruban et ne peut pas se déplacer vers la gauche. Le ruban de sortie quand à lui est prévu pour décrire un nombre réel y de la même manière, par une suite de rationnels $\langle q'_i \rangle$ qui le définit. Sur le ruban de sortie, lorsque la tête écrit un symbole, elle se déplace vers la droite sur le prochain symbole blanc. Sur une entrée, le calcul converge si la machine produit progressivement sa sortie. Si la machine ne produit qu'un nombre fini de symboles ou si la suite ne converge pas comme il faut, le calcul n'est pas défini sur cette entrée.

Définition 7.5. *Une fonction $f : \mathbb{R} \rightharpoonup \mathbb{R}$ est calculable s'il existe une machine de Turing de type 2 qui à toute suite de rationnels (q_i) définissant un nombre réel x associe une suite de rationnels (q'_i) définissant le réel $f(x)$.*

Théorème 7.6. *Toute fonction $f : \mathbb{R} \to \mathbb{R}$ calculable est continue.*

Notes

1. Dans son introduction aux métamathématiques [16], S. Kleene (p. 3) débute son chapitre de théorie des ensembles en mettant en bijection un troupeau de quatre moutons et un bosquet de quatre arbres.

2. Avec deux jarres munies de l'incrémentation, de la décrémentation et du test à zéro, avec un contrôle fini, la bergère n'est ni plus ni moins qu'un automate à deux compteurs dont l'arrêt est indécidable [22] et qui est universelle pour un codage exponentiel de l'entrée dans la première jarre au début du calcul. En ajoutant une troisième jarre, le problème du codage de l'entrée disparaît et l'universalité Turing est complète. Bien sûr, l'utilisation d'un tel système n'est pas réaliste en pratique. Les temps de calcul et les quantités de galets utilisés sont astronomiques et pour atteindre la puissance de calcul

d'un ordinateur contemporain, il faudrait de sacrément grandes jarres ! La bergère a sans doute intérêt à s'occuper de ses moutons plutôt qu'à compiler un document LaTeX codé avec une jarre de galets.

3. Quelle a été l'influence des travaux d'A. Turing sur l'émergence du modèle d'architecture de von Neumann ? Quelle est la contribution de J. von Neumann aux idées présentées dans son rapport, l'EDVAC étant développé par une équipe ? Ch. Babbage et A. Lovelace avaient-ils déjà anticipé les calculateurs à programme stocké en mémoire ? Cette idée était-elle déjà mise en œuvre dans les calculateurs couverts par le secret militaire et mis au point pendant la seconde guerre mondiale ? Pour le propos du présent chapitre, peu importe, les deux articles d'A. Turing et J. von Neumann sont des jalons incontournables (et se lisent très bien même en 2022). Pour le lecteur curieux d'approfondir les aspects historiques, nous conseillons de commencer par la lecture de M. Davis [8] et des articles que B. Randell consacre au sujet [30, 31].

4. La théorie de la complexité se préoccupe de la quantité de ressources nécessaires pour réaliser un calcul, en fonction de la taille de l'entrée. En particulier, il convient d'identifier à quel point cette complexité dépend du modèle de calcul employé. Le lecteur intéressé par une introduction à la théorie de la complexité pourra consulter le manuel de S. Périfel [27] ou celui de S. Arora et B. Barak [1].

5. L'expérience qui consiste à concevoir un processeur complet à partir de portes logiques NON-ET de bascules/verrous est un exercice de codage plutôt satisfaisant. Le lecteur qui voudrait s'y essayer pourrait se tourner vers le cours d'*architecture des ordinateurs* le plus proche, un manuel adapté [26] ou vers des ressources plus ludiques. Sans être exhaustif, l'auteur apprécie : le cours en ligne *From Nand to Tetris* (`https://www.nand2tetris.org/`) et son livre compagnon [23] ; le jeu vidéo minimaliste *MHRD* (`https://www.funghisoft.com/mhrd`) ; le jeu vidéo plus coloré *Turing complete* (`https://turingcomplete.game/`).

6. Les automates de Mealy sont introduits pour la conception de circuits [21]. Ils viennent enrichir, entre autres, les travaux de C. Shannon [33] et D. Huffman [14]. L'étude des automates finis, qui débute avec un fort lien avec d'une part la conception de circuit et d'autre part le modèle de neurones artificiels de W. McCulloch et W. Pitts [20], étudié par S. Kleene [18] qui en tire sa célèbre caractérisation des langages réguliers, deviendra une théorie des automates à part entière. Le lecteur désireux d'en apprendre davantage à ce sujet pourra consulter la retrospective présentée par D. Perrin [28]. On notera que si le présent chapitre adopte un point de vue plutôt circuits booléens, les articles de J. von Neumann [38] et le livre de M. Minsky [22] adoptent encore une présentation sous forme de réseaux de neurones formels dans lequel les délais de propagation restituent directement le caractère séquentiel synchrone sans ajout de bascule/verrou.

7. Il existe de nombreux simulateurs de machines de Turing pour s'essayer à leur programmation. Celui que nous proposons est multi-plateforme `https://nopid.itch.io/gtm` et voici le code source de la machine qui multiplie en binaire, dans le langage de ce simulateur :

```
s, x: fin, _, ->
s, 0: go0, _, ->
s, 1: go1, _, ->

go1: go1, ->
go1, x: ad0, x, ->

gcl: gcl, ->
gcl, _: cl1, _, <-

cl1: cl1, <-
cl1, [0]: cl1, 0, <-
cl1, [1]: cl1, 1, <-
cl1, (0): cl2, (0), ->
cl1, (1): cl2, (1), ->
cl1, _: cl2, _, ->

cl2, 0: cl3, (0), <-
cl2, 1: cl3, (1), <-
cl2, _: cl3, (0), <-

cl3: cl3, <-
cl3, _: cl4, _, <-

cl4: cl4, <-
cl4, [0]: cl4, 0, <-
cl4, [1]: cl4, 1, <-
cl4, _: s, _, ->

ad1, 0: a01, [0], ->
ad1, 1: a10, [1], ->
ad1, _: a01, _, ><

ad0, 0: a00, [0], ->
ad0, 1: A01, [1], ->
ad0, _: gcl, _, ->

a00: a00, ->
a00, _: b00, _, ->

a01: a01, ->
a01, _: b01, _, ->

a10: a10, ->
a10, _: b10, _, ->

b00: b00, ->
b00, 0: rc0, [0], <-
b00, _: rc0, [0], ><
b00, 1: rc0, [1], <-

b01: b01, ->
b01, 0: rc0, [1], <-
b01, _: rc0, [1], ><
b01, 1: rc1, [0], <-

b10: b10, ->
b10, 0: rc1, [0], <-
b10, _: rc1, [0], ><
b10, 1: rc1, [1], <-

rc0: rc0, <-
rc0, _: rr0, _, <-

rr0: rr0, <-
rr0, [0]: ad0, [0], ->
rr0, [1]: ad0, [1], ->

rc1: rc1, <-
rc1, _: rr1, _, <-

rr1: rr1, <-
rr1, [0]: ad1, [0], ->
rr1, [1]: ad1, [1], ->

go0: go0, ->
go0, _: pum, _, ->

pum: pum, ->
pum, 0: bak, (0), <-
pum, 1: bak, (1), <-
pum, _: bak, (0), <-

bak: bak, <-
bak, _: ba2, _, <-

ba2: ba2, <-
ba2, _: s, _, ->

fin, 0: fin, _, ->
fin, 1: fin, _, ->
fin, _: fir, _, ->

fir : fir, ->
fir, _: tra, _, <-

tra: tra, <-
tra, (0): tra, 0, <-
tra, (1): tra, 1, <-
tra, _: OUI, _, ->
```

8. Il existe toute une littérature qui étudie les capacités des machines finies munies de collections finies de galets à sortir de dédales finis ou infinis. Ainsi, il existe d'une part un automate à 2 galets qui explore tout dédale fini et s'arrête et d'autre part un automate à 5 galets capable d'explorer n'importe quel dédale, fini ou infini, en s'arrêtant dans le cas fini. Le lecteur intéressé pourra consulter l'état de l'art [9] de M. Delorme pour un premier tour d'horizon de ces résultats.

9. La thèse de Church-Turing mérite sans doute une discussion plus longue que l'espace restreint accordé ici. Des publications sont régulièrement consacrées à sa formalisation et à ses limites, par exemple [10, 13] qui discutent de son axiomatisation. Pour une présentation plus détaillée du sujet, le lecteur est invité à consulter P. Odifreddi [24] (p.100-123) ou encore B. Copeland [7].

Bibliographie

[1] S. ARORA et B. BARAK : *Computational complexity : a modern approach*. Cambridge University Press, 2009.

[2] P. ARRIGHI et G. DOWEK : The physical church-turing thesis and the principles of quantum theory. *International Journal of Foundations of Computer Science*, 23(05):1131–1145, 2012.

[3] O. CARTON : *Langages formels, calculabilité et complexité*. Vuibert, 2008.

[4] A. CHURCH : An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58(2):345–363, 1936.

[5] J. COCKE et M. MINSKY : Universality of tag systems with p= 2. *Journal of the ACM (JACM)*, 11(1):15–20, 1964.

[6] M. COOK *et al.* : Universality in elementary cellular automata. *Complex systems*, 15(1):1–40, 2004.

[7] B. J. COPELAND : The Church-Turing Thesis. *Dans* E. N. ZALTA, éd. : *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, Summer 2020 édn, 2020.

[8] M. D. DAVIS : *The Universal Computer : The Road from Leibniz to Turing*. W. W. Norton & Company, 2000.

[9] M. DELORME : Automates à galets : un état de l'art. Research Report LIP RR-1997-23, Laboratoire de l'informatique du parallélisme, août 1997. URL `https://hal-lara.archives-ouvertes.fr/hal-02101923`.

[10] N. DERSHOWITZ et Y. GUREVICH : A natural axiomatization of computability and proof of church's thesis. *The Bulletin of Symbolic Logic*, 14(3):299–350, 2008.

[11] A. DOXIADIS et C. PAPADIMITRIOU : *LOGICOMIX : an epic search for truth*. Bloomsbury Publishing USA, 2009.

[12] R. GANDY : Church's thesis and principles for mechanisms. *Dans Studies in Logic and the Foundations of Mathematics*, vol. 101, pp. 123–148. Elsevier, 1980.

[13] Y. GUREVICH *et al.* : Unconstrained church-turing thesis cannot possibly be true. *Bulletin of EATCS*, 1(127), 2019.

[14] D. A. HUFFMAN : The synthesis of sequential switching circuits. *Journal of the franklin Institute*, 257(3):161–190, 1954.

[15] N. D. JONES : *Computability and complexity : from a programming perspective*, vol. 21. MIT press, 1997.

[16] S. KLEENE : *Introduction to Metamathematics*. Bibliotheca mathematica. Van Nostrand, 1952.

[17] S. C. KLEENE : General recursive functions of natural numbers. *Mathematische annalen*, 112(1):727–742, 1936.

[18] S. C. KLEENE : Representation of events in nerve nets and finite automata. *Dans* C. E. SHANNON et J. MCCARTHY, éds : *Automata Studies*, pp. 3–41. Princeton University Press, Princeton, 1956.

[19] A. A. MARKOV : The theory of algorithms. *Trudy Matematicheskogo Instituta Imeni VA Steklova*, 42:3–375, 1954.

[20] W. S. MCCULLOCH et W. PITTS : A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, 5:115–133, 1943.

[21] G. H. MEALY : A method for synthesizing sequential circuits. *The Bell System Technical Journal*, 34(5):1045–1079, 1955.

[22] M. MINSKY : *Computation : Finite and Infinite Machines*. Prentice Hall, Englewoods Cliffs, 1967.

[23] N. NISAN et S. SCHOCKEN : *The Elements of Computing Systems : Building a Modern Computer from First Principles*. MIT Press, 2005.

[24] P. ODIFREDDI : *Classical recursion theory : The theory of functions and sets of natural numbers*. Elsevier, 1992.

[25] N. OLLINGER : Universalities in cellular automata. *Dans* G. ROZENBERG, T. BÄCK et J. N. KOK, éds : *Handbook of Natural Computing*, pp. 189–229. Springer, Berlin, 2012.

[26] D. A. PATTERSON et J. L. HENNESSY : *Computer Organization and Design*. Morgan Kaufmann Publishers, 2007.

[27] S. PERIFEL : *Complexité algorithmique*. Ellipses, 2014. URL `https://www.irif.fr/users/sperifel/livre_complexite`.

[28] D. PERRIN : Les débuts de la théorie des automates. *Technique et Science Informatique*, 14:409–433, 1995.

[29] E. POST : Formal reductions of the general combinatorial decision problem. *American Journal of Mathematics*, 65(2):197–215, 1943.

[30] B. RANDELL : On Alan Turing and the origins of digital computers. *Computing Laboratory Technical Report Series*, 1972.

[31] B. RANDELL : The history of digital computers. *Bulletin of the Institute of Mathematics and its Applications*, 1976.

[32] H. G. RICE : Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical society*, 74(2):358–366, 1953.

[33] C. E. SHANNON : The synthesis of two-terminal switching circuits. *The Bell System Technical Journal*, 28(1):59–98, 1949.

[34] M. SIPSER : *Introduction to the Theory of Computation*. Course Technology, 2006.

[35] R. I. SOARE : The history and concept of computability. *Handbook of computability theory*, 140:3–36, 1999.

[36] A. M. TURING : On computable numbers with an application to the entscheidungs problem. *Proceedings of the London Mathematical Society 2*, 42:230–265, 1936.

[37] A. M. TURING : Computability and λ-definability. *The Journal of Symbolic Logic*, 2(4):153–163, 1937.

[38] J. von NEUMANN : First draft of a report on the EDVAC. Rap. tech., Moore School of Electrical Engineering, University of Pennsylvania, 1945. (doi :10.5479/sil.538961.39088011475779).

[39] K. WEIHRAUCH : *Computable analysis : an introduction*. Springer Science & Business Media, 2000.

[40] D. WOODS et T. NEARY : The complexity of small universal Turing machines. *Actes de Computability in Europe 2007*, vol. 4497 de *LNCS*, pp. 791–798. Springer, 2007.