Théorème de récursion de Kleene

Document rédigé pour le site CultureMath

Ghazal Kachigar

8 mai 2017

Table des matières

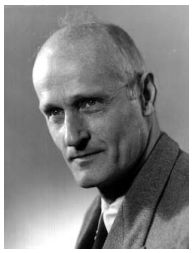
1 Introduction

De nos jours, nous sommes entourés d'ordinateurs : des ordinateurs dont tout le monde est d'accord pour dire que ce sont des ordinateurs, mais aussi les smartphones, par exemple. Cela pourrait choquer les personnes qui ne se sont pas intéressées au problème de définition de ce qu'est un ordinateur. Mais un ordinateur est simplement un objet physique capable d'implémenter tout ce qui est calculable, et la question mathématique de savoir ce qui est calculable a précédé et même alimenté le développement de ces ordinateurs physiques. En cherchant à répondre à ces questions, les mathématiciens ont découvert des propriétés étonnantes vérifiées par ces ordinateurs théoriques et donc aussi leurs implémentations physiques, alors même que ces dernières n'existaient pas encore ou commençaient à paraître.

Un résultat fondamental de ce domaine de recherche, appelé **théorie de la calculabilité** ou encore **théorie de la récursion**, est le **théorème de récursion de Kleene** et ses variantes.

Le but de ce texte est d'expliquer, dans un premier temps de manière intuitive, la signification de ce théorème et quelques-unes de ses applications.

Dans un deuxième temps, nous définirons et prouverons tout ce qui a été dit de manière formelle. Nous verrons notamment que, bien que les preuves soient très abstraites, ils sont d'une simplicité étonnante dans leur forme et par les arguments qu'ils mobilisent.



Stephen Cole Kleene (1909-1994)

2 Comprendre le théorème de récursion de Kleene

2.1 De quoi est fait un ordinateur ?

Vous utilisez votre ordinateur pour effectuer diverses tâches. Cela passe par l'exécution de **programmes** ou **logiciels**. Le plus important de ces programmes est votre **système d'exploitation** qui se lance peu après que l'ordinateur est allumé. Que fait le système d'exploitation ? Dans des conditions d'exécution normales, il **tourne jusqu'à ce que** vous lui donniez la commande d'éteindre l'ordinateur. Pendant qu'il tourne, vous pouvez lui demander de lancer des programmes qui sont installés sur le disque dur : un logiciel de traitement de texte, un navigateur web, un jeu vidéo, entre autres. Il peut faire cela car quelque part sur votre disque dur, il y a le **code machine** de ce programme, qui fournit

au système d'exploitation une série d'instructions qui permettent, à lui qui est lui-même un programme, d'exécuter un autre programme. D'ailleurs, on peut soi-même écrire des programmes informatiques, dans un **langage de programmation**, et faire appel à un programme appelé compilateur pour les traduire en code machine.

Nous allons utiliser ce qui a été dit afin de dégager quelques notions fondamentales en informatique.

Un **programme** est la réalisation, sur un ordinateur physique, d'une fonction calculable ou un algorithme. Il peut donc avoir un ou plusieurs arguments, ses **entrées**, auxquels il associe une valeur, sa **sortie**. Nous pouvons appeler cela **son comportement sémantique**. On peut lui associer un **code** dans un langage de programmation, ce que nous pouvons voir comme sa **syntaxe**. Un programme suffisamment sophistiqué peut exécuter un autre programme à partir de son code, autrement dit, prendre en entrée une donnée syntaxique qui est le code du programme en question et imiter son comportement sémantique. À noter que ceci est distinct d'un programme qui ferait des tâches pouvant donner lieu en elle-mêmes à des programmes à part entière, par exemple un programme de multiplication basée sur la répétition de l'addition.

Un programme comporte des **structures de contrôle**. Les **alternatives** sont des instructions dont l'exécution dépend du remplissement ou non d'une condition (si A, alors faire B, sinon faire C). Les **boucles** sont des instructions qui sont répétées, ou bien un nombre de fois fini et prédéterminé (boucle **pour** ou **for**), ou bien tant qu'une condition est remplie (boucle **tant que** ou **while**), à l'instar du système d'exploitation qui continue à tourner tant qu'il n'y a pas la commande d'éteindre l'ordinateur. Or, nous verrons que pour avoir les boucles while qui terminent, il faut également admettre la possibilité des boucles infinies, où la condition de terminaison de la boucle n'est jamais remplie. Par conséquent, l'incorporation des boucles while introduit une rupture fondamentale. Même si du point de vue pratique, on essaie d'éviter les boucles infinies et il y a toujours la possibilité de couper le courant, il faut garder à l'esprit que sans les boucles while, le système d'exploitation, ce programme qui exécute bien d'autres programmes à partir de leur code, n'aurait pas été possible.

Pour résumer, donc, il y a deux types de programmes : les programmes qui ne comportent pas de structure de contrôle de type boucle while, que nous appellerons les **programmes simples**, et les programmes qui comportent des structures de contrôle de type boucle while, que nous appellerons les **programmes conditionnels**. Nous pouvons aussi distinguer, parmi les programmes conditionnels, ceux qui se terminent et ceux qui comportent une boucle infinie (le vocabulaire scientifique correct sera introduit dans la partie mathématique).

2.2 Théorème de récursion de Kleene

Ce qui a été dit précédemment est intéressant en soi, mais également important pour saisir la signification des résultats que nous allons étudier de près. Nous allons donc introduire quelques notations pour pouvoir énoncer les résultats de manière succincte.

Fixons un système de codage (code machine ou un langage de programmation) : chaque programme a un code dans ce système. Nous allons alors utiliser la notation ϕ_e pour désigner le programme qui a pour code e dans ce système de codage. Le code e est par conséquent une donnée syntaxique associée à ce programme. Un programme f peut avoir une ou plusieurs entrées : nous écrirons $f(x_1, ..., x_n)$ pour désigner le déroulement du programme f, donc son comportement sémantique, sur les entrées $x_1, ..., x_n$. Ainsi, la notation $\phi_e(x)$ sera utilisée pour désigner le déroulement du programme ayant pour code e dans le système de codage fixé sur l'entrée x.

Nous allons maintenant énoncer et commenter deux formes du théorème de récursion de Kleene.

2.2.1 Théorème de récursion de Kleene, première forme

Pour tout programme conditionnel f, il existe un programme ayant pour code e tel que $\phi_e(x) = f(e, x)$ pour n'importe quelle entrée x.

Il s'agit d'une formulation proche de la formulation initiale du théorème par Kleene en 1938. Ce qui est important à noter dans cette formulation est que e est un code et joue donc un rôle syntaxique à gauche alors qu'il est un argument d'une fonction et joue donc un rôle sémantique à droite. Nous avons vu qu'il n'y a rien d'étonnant à ce que le code d'un programme soit un argument d'entrée d'une autre fonction. Ce qui est étonnant ici est que le code de la fonction en question est identique à l'un de ses arguments.

2.2.2 Théorème de récursion de Kleene, deuxième forme

Pour tout programme conditionnel h qui se termine, il existe un programme conditionnel ayant pour code e tel que $\phi_e = \phi_{h(e)}$, i.e. pour n'importe quelle entrée x, $\phi_e(x) = \phi_{h(e)}(x)$.

Cette forme du théorème de récursion de Kleene est également appelé **théorème de point fixe de Rogers** [5]. En effet, e est un point fixe *sémantique* du programme (de la fonction) h dans la mesure où le théorème affirme que les programmes ayant pour codes e et $h(e)$ ont le même comportement.

Hartley Rogers, Jr. (1926-2015)

Notons également que le théorème est également cohérent au niveau de la forme : nous avons dit précédemment que le code d'un programme peut tout

à fait servir comme l'entrée d'un autre programme. De plus, puisque h est un programme qui se termine, la valeur $h(e)$ est bien définie.

2.2.3 Résultats voisins et autres formes

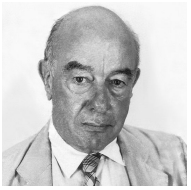
Nous verrons dans la partie mathématique que la preuve des deux formes du théorème de récursion de Kleene se base sur un autre théorème fondamental appelé **théorème d'itération** ou **théorème s-m-n** et qui est également dû à Kleene. Une partie de ce qu'affirme ce théorème semble aller de soi : s'il existe un programme à $m+n$ arguments, alors on obtient un autre programme en fixant ses m premiers arguments. Par exemple, en fixant le premier argument d'un programme qui prend en entrée x et y et qui calcule $x+y$, on obtient d'autres programmes : par exemple, en fixant x à 5, on obtient le programme qui prend en entrée y et qui calcule $5+y$. Ce théorème affirme toutefois quelque chose en plus, à savoir que le code du deuxième programme peut être calculé en appliquant un programme simple au code du premier programme et aux m arguments fixés.

Notons également qu'il y a des formes plus générales du théorème de récursion de Kleene, mais aussi des théorèmes de récursion doubles avec deux programmes e_1 et e_2 au lieu d'un seul. Mais nous allons principalement nous intéresser aux deux formes énoncées ci-dessus, qui permettent déjà d'obtenir des résultats non-triviaux.

2.3 Deux conséquences

2.3.1 Les programmes autoréplicateurs

On appelle **autoréplicateur** un programme dont l'entrée n'est pas son code mais dont au moins une partie de la sortie est son code. Lorsque la sortie est exactement le code, un tel programme s'appelle un **quine**, en l'honneur du philosophe et logicien américain du même nom qui a beaucoup travaillé sur l'auto-référence. Il faut peu d'hypothèses en plus pour déduire qu'il existe des quines non-triviaux (i.e. différent du programme vide) à partir des deux formes du théorème de récursion énoncé ci-dessus. En effet, un quine est un programme ayant pour code e et pour sortie e, autrement dit, $\phi_e(x) = e$ pour tout x.



Willard Van Orman Quine (1908-2000)

Avec la première forme, il suffit de prendre la fonction $f(e,x) := e$ pour tout x pour obtenir ce résultat.
Avec la deuxième forme, il suffit de prendre h tel que $\phi_{h(e)}(x) = e$ pour tout x.

Il faudrait encore se convaincre que ces fonctions sont bien calculables, mais nous verrons que c'est bien le cas dans la partie mathématique de ce texte.

La page http ://rosettacode.org/wiki/Quine regroupe des codes sources de quines dans une multitude de langages de programmation.

Il existe d'autres formes, moins drôles, de programmes autoréplicateurs : ce sont les virus et les vers informatiques. Ces programmes font partie de la famille plus grande des codes et logiciels malveillants. De nos jours, le moyen de lutte le plus efficace contre ce type de logiciels repose sur la technique de **recherche de signature**, c'est-à-dire la recherche de motifs caractéristiques dans le code du logiciel. Mais cette technique a ses limites car les logiciels pourraient être équipés pour modifier ou obfusquer leurs codes. Il y a donc des tentatives d'approcher ce problème autrement, en essayant de caractériser les vers et les virus informatiques à l'aide du théorème de récursion et ses variantes [2].

2.3.2 Le théorème de Rice sur les propriétés sémantiques des programmes

Rappelons qu'un programme a un code, qui est une donnée syntaxique, et lorsqu'il est déroulé sur une entrée, il a un comportement sémantique. On peut donc s'intéresser à la question suivante : est-il possible d'écrire un programme capable de déterminer quels sont les programmes qui ont un comportement sémantique particulier ?

Par exemple, étant donné un argument x, pouvons-nous déterminer, à l'aide d'un programme, l'ensemble des programmes conditionnels qui se terminent sur l'entrée x ? Ce problème, appelé **le problème de l'arrêt**, est une question intéressante en informatique mais aussi en mathématiques. En effet, il y a des problèmes mathématiques non-résolus qui demandent si un objet vérifiant une propriété particulière existe. Si l'on peut écrire un programme qui parcourt les objets en question jusqu'à en trouver un qui vérifie la propriété, il suffirait alors de regarder si ce programme figure parmi les programmes qui se terminent pour pouvoir répondre à la question.

Il y a d'autres propriétés, telle celle de savoir si un programme termine sur n'importe quelle entrée x (le **problème de la totalité**) ou bien celle de savoir si deux programmes font la même chose (le **problème de l'équivalence**).

Le théorème de Rice (1953) permet de répondre à cette question importante par la négative : si un comportement sémantique n'est pas **trivial**, c'est-à-dire partagé par tous les programmes ou par aucun programme, le théorème de Rice affirme qu'il n'existe aucun programme permettant de déterminer l'ensemble des programmes ayant ce comportement sémantique.

Nous allons donner les idées derrière la preuve de ce théorème qui n'est pas aussi immédiate que celle de l'existence des quines : on procède par contradiction, en supposant qu'un tel programme existe (hypothèse H1) et on considère un ensemble A de programmes ayant un comportement sémantique donné. La propriété suivante, que l'on notera (*), sera utile par la suite :

(*) Si deux programmes ϕ_e et $\phi_{e'}$ ont le même comportement sémantique, ils

sont tous les deux soit dans A ou en dehors de A.

Nous considérons alors un programme ϕ_n qui est dans A et un programme ϕ_m qui est en dehors de A. Ensuite, nous considérons le programme h qui prend en entrée x et y et qui déroule $\phi_m(y)$ si ϕ_x est dans A et $\phi_n(y)$ sinon.

Nous appliquons ensuite la première forme du théorème de récursion de Kleene : il existe e tel que $h(e, y) = \phi_e(y)$ pour tout y.

Nous pouvons alors poser la question suivante au sujet de ϕ_e : est-ce que ϕ_e est dans A ?

Si c'est le cas (hypothèse H2), par définition de e nous avons $\phi_e(y) = h(e, y)$ et par définition de h, $h(e, y) = \phi_m(y)$. Par conséquent, $\phi_e(y) = \phi_m(y)$ pour tout y. Par conséquent, ϕ_e et ϕ_m ont le même comportement sémantique et par (*), ils doivent être tous les deux dans A ou en dehors de A. Mais ϕ_e est dans A alors que ϕ_m non. On aboutit donc à une contradiction, ce qui signifie que l'hypothèse H2 doit être fausse.

Il reste donc à voir ce qui se passe lorsque ϕ_e n'est pas dans A. Alors, par définition de e $\phi_e(y) = h(e, y)$ et par définition de h, $h(e, y) = \phi_n(y)$ et donc $\phi_e(y) = \phi_n(y)$. On retrouve le même raisonnement : par (*), on devrait avoir ϕ_e et ϕ_n à la fois dans A ou en dehors de A. Il y a donc une fois de plus une contradiction, ce qui signifie que c'est l'hypothèse que nous avions fait plus haut, H1, qui est fausse, ce qui achève la démonstration du théorème.

3 Formulation mathématique

Dans cette section, nous allons définir et prouver tout ce qui a été dit mathématiquement.

Nous présenterons dans une première partie la notion de **fonction récursive** qui capture formellement ce qu'est un programme informatique. Cela passe par l'introduction des notions de **fonction primitive-récursive** et **fonction μ-récursive** qui représentent ce que nous avons appelé respectivement un programme simple et un programme conditionnel. Nous en énoncerons ensuite quelques propriétés, notamment la possibilité d'associer un code à chaque fonction récursive via **la numérotation de Gödel**.

Dans une deuxième partie, nous commencerons par présenter le **théorème d'itération**, ce qui nous donnera tous les outils nécessaires pour prouver les deux formes du théorème de récursion de Kleene. Nous conclurons par les deux corollaires de ce théorème, à savoir l'existence de **fonctions auto-réplicatrices** et le **théorème de Rice**.

3.1 Une définition de la calculabilité

Il y a **plusieurs approches** pour définir la calculabilité. Ces approches sont toutes équivalentes, mais chacune d'entre elles met l'accent sur un aspect

différent de la question. Nous optons ici pour l'approche basée sur les fonctions récursives, car elle est la plus simple pour pouvoir comprendre la dichotomie syntaxe/sémantique qui a été mentionnée à plusieurs reprises.
Nous allons considérer des fonctions qui prennent en entrée des arguments dans $\mathbb{N}^k$ pour $k \in \mathbb{N}$.[1] Dans un souci de concision d'écriture, nous écrirons $\vec{x}$ au lieu de $(x_1, ..., x_k)$ pour les arguments d'une telle fonction.

3.1.1 Fonctions récursives

Rappelons la classification des programmes en programmes simples et programmes conditionnels. Nous allons maintenant les définir rigoureusement et en donner les appellations mathématiques.

Nous allons commencer par définir deux opérations.

Définition 1 (Composition). *Étant donné $k, j \in \mathbb{N}$,*
des fonctions $g : \mathbb{N}^k \to \mathbb{N}$, $h_1, ..., h_k : \mathbb{N}^j \to \mathbb{N}$
nous définissions la composition $f : \mathbb{N}^j \to \mathbb{N}$ de g avec $h_1, ..., h_j$ comme suit :
$f(\vec{n}) := g(h_1(\vec{n}), ..., h_k(\vec{n}))$

Définition 2 (Récursion primitive). *Étant donné $k \in \mathbb{N}$,*
des fonctions $g : \mathbb{N}^k \to \mathbb{N}$, $h : \mathbb{N}^{k+2} \to \mathbb{N}$
nous définissions la récursion primitive $f : \mathbb{N}^{k+1} \to \mathbb{N}$ de g avec h comme suit :
$f(0, \vec{n}) := g(\vec{n})$
$f(m+1, \vec{n}) := h(f(m, \vec{n}), \vec{n}, m)$

Cette définition peu intuitive correspond en réalité à une boucle for, c'est-à-dire une suite d'instruction répétées pendant un nombre fini et prédéterminé d'étapes.

Nous pouvons maintenant définir l'ensemble des fonctions primitives-récursives par induction, c'est-à-dire en donnant des objets de base qui y appartiennent et des opérations qui permettent d'en construire d'autres en partant de ces objets de base.

Définition 3 (Fonctions primitives-récursives). *L'ensemble des fonctions primitives-récursives, noté* Prim*, est composé des fonctions suivantes :*

1) La fonction constante égale à 0 : $C(n_1, ..., n_k) = 0$.
2) La fonction successeur : $S(n) = n + 1$.
3) Les fonctions de projection : $P_i(n_1, ...n_k) = n_i$ pour $1 \leq i \leq k$

Les autres membres de Prim *sont les fonctions f construites par les opérations de :*

a) composition
b) récursion primitive

à partir de fonctions g, h, $h_1, ..., h_k \in$ Prim.

1. Comme nous le verrons plus tard, le domaine de définition de ces fonctions peut être $\mathbb{N}^k$ tout entier ou bien un sous-ensemble propre de $\mathbb{N}^k$.

En d'autres termes, PRIM est le plus petit ensemble contenant la fonction constante nulle, la fonction successeur, les fonctions de projection et qui est clos pour les opérations de composition et de récursion primitive.

Ainsi, les programmes que nous avons appelés programmes simples correspondent aux fonctions primitives-récursives. Nous allons donner quelques exemples pour nous familiariser avec ces fonctions.

EXEMPLE 1 (Les fonctions constantes). *On peut montrer que les fonctions constantes $C_\ell(n_1, ..., n_k) = \ell$ sont primitives récursives. En effet, nous savons que la fonction constante nulle C_0 est primitive-récursive, ainsi que la fonction successeur S, et que l'opération de composition permet d'obtenir d'autres fonctions primitives-récursives. On peut alors montrer par récurrence, en utilisant l'égalité $C_\ell(n_1, ..., n_k) = S(C_{\ell-1}(n_1, ..., n_k))$, que $C_\ell \in$ PRIM.*

EXEMPLE 2 (La somme de deux entiers). *On peut montrer que la fonction $f(x, y) := x + y$ est primitive récursive. En effet, nous avons montré que la fonction C_y est primitive-récursive. On peut alors utiliser C_y, S, P_1 et les opérations de composition et de récursion primitive pour définir f :*
$f(0, y) = C_y()$
$f(x + 1, y) = S(P_1(f(x, y), y, x)$
Cela revient à faire la chose suivante : si $x = 0$, le résultat est y. Sinon, on ajoute 1 au résultat de la fonction f appliquée à $(x - 1, y)$, et ainsi de suite jusqu'à arriver à $(0, y)$. Ainsi, cela revient à prendre comme valeur de départ y et répéter x fois l'opération «ajouter 1».

Notons également que qu'une fonction primitives-récursive k-aire est définie sur l'ensemble de $\mathbb{N}^k$: on dit d'une telle fonction qu'elle est **totale**. Nous voulons maintenant donner une définition mathématique des programmes conditionnels. Pour ce faire, nous introduisons l'opérateur μ qui correspond à la boucle while.

DÉFINITION 4 (Opérateur μ). *Étant donné une fonction totale $f : \mathbb{N}^{k+1} \to \mathbb{N}$, on définit l'opérateur de minimisation ou l'opérateur μ de la manière suivante :*

$\mu(f)(n_1, ..., n_k) = m$ *si et seulement si :*
$f(i, n_1, ..., n_k) > 0$ *pour* $1 \leq i \leq m - 1$ *et* $f(m, n_1, ..., n_k) = 0$

Une formulation équivalente :

$\mu(f)(n_1, ..., n_k) = \min\{m \in \mathbb{N} : f(i, n_1, ..., n_k) > 0$ *pour* $1 \leq i \leq m - 1$ *et* $f(m, n_1, ..., n_k) = 0\}$

Or, il est tout à fait possible qu'il n'y ait aucun $m \in \mathbb{N}$ pour lequel $f(m, n_1, ..., n_k) = 0$. Dans ce cas, l'opérateur μ n'est pas défini pour l'argument $(n_1, ..., n_k)$. La fonction obtenue en appliquant l'opérateur μ n'est alors pas totale : son domaine de définition n'est pas $\mathbb{N}^k$ tout entier. On dit qu'elle est **partielle**.

DÉFINITION 5 (Fonctions μ-récursives). *L'ensemble* REC *des fonctions μ-récursives est le plus petit ensemble contenant la fonction constante nulle, la fonction successeur, les fonctions de projection et qui est clos pour les opérations de composition, récursion primitive et l'opérateur μ.*

Nous allons maintenant formaliser les notions de fonctions totales et partielles, qui correspondent à des programmes qui terminent pour n'importe quelle entrée et des programmes qui peuvent boucler à l'infini sur certaines entrées.

NOTATION 1. *Soit* $f : \mathbb{N}^k \to \mathbb{N} \in$ REC.

1. *Si* f *est définie en* $\vec{n}$*, c'est-à-dire s'il existe* $m \in \mathbb{N}$ *tel que* $f(\vec{n}) = m$*, nous écrirons* $f(\vec{n}) \downarrow$.
2. *Sinon (si* $\vec{n}$ *n'est pas dans le domaine de définition de* f*) nous écrirons* $f(\vec{n}) \uparrow$.

DÉFINITION 6 (Fonctions totales et partielles). *Soit* $f : \mathbb{N}^k \to \mathbb{N} \in$ REC.

1. *Si pour tout* $\vec{n} \in \mathbb{N}^k$*, on a* $f(\vec{n}) \downarrow$*, on dit que* f *est totale.*
2. *Sinon (si le domaine de définition de* f *est un sous-ensemble propre de* $\mathbb{N}^k$*), on dit que* f *est partielle.*

Nous sommes maintenant en mesure de donner une définition mathématique de l'idée d'avoir le même comportement sémantique.

DÉFINITION 7 (Équivalence). *Soit* $f, g : \mathbb{N}^k \to \mathbb{N} \in$ REC*. On dit que* f *et* g *sont équivalentes, et on écrit* $f \simeq g$*, si :*

1. $f(\vec{n}) \downarrow$ *si et seulement si* $g(\vec{n}) \downarrow$.
2. *Pour tous les* $\vec{n} \in \mathbb{N}^k$ *tels que* $f(\vec{n}) \downarrow$*,* $f(\vec{n}) = g(\vec{n})$.

Nous avons donc donné des définitions mathématiques des trois types de programmes : les programmes simples correspondent aux fonctions primitives-récursives, les programmes conditionnels qui terminent toujours aux fonctions μ-récursives totales et les programmes conditionnels sans fin pour certaines entrées, aux fonctions μ-récursives partielles.

Quel rapport entre tout cela et la calculabilité ? Il est un peu fastidieux, mais faisable, de calculer chaque fonction μ-récursive à l'aide de sa construction à partir des fonctions de base : ces fonctions sont donc intuitivement calculables. La **thèse de Church-Turing** affirme que la notion intuitive de calculabilité est capturée par la notion de μ-récursivité. Cette thèse a un statut de postulat ou de conjecture et non de théorème, car elle fait le lien entre une notion non-mathématique et une notion mathématique.

Alonzo Church (1903-1995)

Alan Turing (1912-1954)

3.1.2 Existence d'un codage

Dans cette section, nous allons démontrer la possibilité d'associer à chaque fonction μ-récursive un entier naturel qui serait son code, c'est-à-dire l'existence d'une bijection entre l'ensemble des fonctions μ-récursives et un sous-ensemble propre des entiers naturels. Il y a plusieurs manières de procéder, nous allons présenter un grand classique.

Une démonstration complète de ce résultat serait trop longue. Nous allons donc admettre certains résultats intermédiaires dont on peut trouver la preuve ailleurs ou en guise d'exercice pour s'approprier les concepts définis dans la partie précédente.

LEMME 1. *Les fonctions suivantes sont primitives-récursives.*

1. $moins(m,n) = 0$ *si* $m < n$, $m - n$ *sinon.*
2. $mult(x_1, ..., x_k) = x_1 * ... * x_k \ \forall k \in \mathbb{N}$.
3. $exp(n,m) = n^m$.
4. $v(n,p) = i$, *où* i *est le plus grand entier tel que* p^i *divise* n.
5. $premier(n) = p_n$ *le* n*ème plus petit nombre premier.*
6. $longueur(x_1, ..., x_k) = k$.

Nous aurons également besoin du **théorème fondamental de l'arithmétique**.

THÉORÈME 1 (Théorème fondamental de l'arithmétique). *Tout entier strictement positif admet une décomposition unique en produit de facteurs premiers.*

Nous pouvons maintenant énoncer et démontrer l'existence d'un codage des fonctions μ-récursives en entiers naturels.

THÉORÈME 2 (Existence d'un codage). *Il existe une bijection primitive-récursive entre la classe* REC *et un sous-ensemble* $\mathbb{G} \subsetneq \mathbb{N}$.

Démonstration. Nous allons présenter ici l'idée de la **numérotation de Gödel** (d'où le symbole $\mathbb{G}$).

Nous notons d'abord qu'il existe une manière naturelle de «coder» les fonctions μ-récursives en uplets, en représentant dans l'ordre les informations importantes qui s'y rapportent. Pour les fonctions de base, il y a d'abord leur type (fonction nulle, successeur ou projecteur), ensuite leur arité, et pour les projecteurs, la

coordonnée sur laquelle on projette.

Nous pouvons donc définir une fonction U qui associe à chacune des fonctions de base un uplet, de la manière suivante :

Fonction nulle k-aire : $U(C) = (1, k)$
Fonction successeur unaire : $U(S) = (2, 1)$. En effet, S est d'arité 1.
Fonction de projection k-aire : $U(P_i) = (3, k, i)$

Il faut maintenant étendre la fonction U aux trois opérations (composition, récursion primitive et opérateur μ) qui permettent de construire de nouvelles fonctions à partir de celles-ci. Pour cela, nous avons besoin de savoir de quelle opération il s'agit, ainsi que des informations sur les fonctions utilisées.

Composition de g, k-aire, avec $h_1, ..., h_k$:
$U(Comp, g, h_1, ..., h_k) = (4, k, U(g), U(h_1), ..., U(h_k))$
Récursion primitive $(k+1)$-aire de g k-aire avec h :
$U(RecPrim, g, h) = (5, k+1, U(g), U(h))$
Opérateur μ k-aire sur f $(k+1)$-aire :
$U(\mu, f) = (6, k, U(f))$

Il est facile de se convaincre que ce codage en uplets est univoque pour chaque fonction. Nous pouvons donc identifier une fonction et le uplet qui la représente. Nous allons maintenant donner une fonction $\mathcal{G}$ qui associe un entier naturel unique à chaque uplet.

Cette fonction agit sur les couples et triplets associés aux fonctions de base de la manière suivante :

$$\begin{aligned}
\mathcal{G} : (u_1, u_2) &\mapsto mult(exp(premier(1), u_1 + 1), exp(premier(2), u_2 + 1)) \\
&\mapsto p_1^{u_1+1} p_2^{u_2+1} \\
&\mapsto 2^{u_1+1} 3^{u_2+1} \\
\mathcal{G} : (u_1, u_2, u_3) &\mapsto p_1^{u_1+1} p_2^{u_2+1} p_3^{u_3+1} \\
&\mapsto 2^{u_1+1} 3^{u_2+1} 5^{u_3+1}
\end{aligned}$$

Nous voyons que $\mathcal{G}$ est construite par composition à partir de certaines fonctions mentionnées dans le Lemme 1. Elle est par conséquent primitive récursive.

Pour les fonctions plus complexes, $\mathcal{G}$ est définie de manière analogue, à cette différence près : lorsqu'un élément du uplet est lui-même un uplet (car il coderait une fonction) et non un entier, la fonction $\mathcal{G}$ fait appel à elle-même sur cet uplet. Illustration de cette procédure pour la récursion primitive, $(5, k+1, U(g), U(h))$:

$$\begin{aligned}
\mathcal{G} : (4, k+1, U(g), U(h)) &\mapsto p_1^{4+1} p_2^{(k+1)+1} p_3^{\mathcal{G}(U(g))+1} p_4^{\mathcal{G}(U(h))+1} \\
&\mapsto 2^5 3^{k+2} 5^{\mathcal{G}(U(g))+1} 7^{\mathcal{G}(U(h))+1}
\end{aligned}$$

Notons que les fonctions g et h peuvent être des fonctions de base ou des fonctions complexes, auquel cas il y aura encore des appels à $\mathcal{G}$. Mais au bout d'un

certain nombre d'appels, on arrivera aux fonctions de base. Nous pouvons alors montrer par récurrence, en utilisant le fait que la fonction $\mathcal{G}$ définie sur les fonctions de bases est primitive-récursive, que son extension définie de cette manière reste primitive-récursive.

Il est clair que $\mathcal{G}$ est injective et définit donc une bijection $\text{Rec} \mapsto Im(\mathcal{G}) := \mathbb{G}$.

Pour terminer la preuve, il reste à montrer que la fonction inverse $\mathcal{G}^{-1}$ est primitive-récursive et bijective. Notons, pour cela, que $\mathcal{G}^{-1}$ se calcule de la manière suivante pour les fonctions de base, par exemple les projecteurs :

$$\begin{aligned}\mathcal{G}^{-1} : n = p_1^{k_1} p_2^{k_2} p_3^{k_3} &\mapsto (k_1 - 1, k_2 - 1, k_3 - 1) \\ &\mapsto (v(n, p_1) - 1, v(n, p_2) - 1, v(n, p_3) - 1)\end{aligned}$$

On y trouve des fonctions mentionnées dans le Lemme 1 et l'opération de composition. Par conséquent, cette fonction est primitive-récursive.

Il est possible étendre $\mathcal{G}^{-1}$ à l'ensemble des fonctions μ-récursives de manière similaire à ce qui a été fait pour $\mathcal{G}$ et montrer qu'elle est primitive-récursive.

Enfin, le fait que $\mathcal{G}^{-1}$ soit bijective résulte du théorème fondamental de l'arithmétique. ■

Kurt Gödel (1906-1978)

Définition 8 (Code d'une fonction). *Nous appelons l'image n d'une fonction f par le codage définie ci-dessus son* ***code****, et nous utiliserons la notation ϕ_n pour la fonction ayant pour code n dans ce système de codage.*

3.1.3 Des fonctions aux ensembles : la décidabilité

Dans cette section, nous allons transposer la notion de calculabilité aux ensembles. Nous ferons appel à ces notions principalement dans la partie 3.2.4 sur le théorème de Rice.

Notation 2 (Fonction caractéristique). *Étant donné un ensemble (ou une relation) $A \subset \mathbb{N}^k$, nous noterons χ_A sa fonction caractéristique, i.e. la fonction à valeurs dans $\{0, 1\}$ telle que, pour tout $x \in \mathbb{N}^k$ $\chi_A(x) = 1$ si et seulement si $x \in A$.*

Définition 9 (Ensemble décidable). *Nous disons qu'un ensemble $A \subset \mathbb{N}^k$ est* ***décidable*** *si χ_A est μ-récursive et totale.*

LEMME 2 (Définition par cas). *Soient A_1, ... , A_m des ensembles décidables formant une partition de $\mathbb{N}^k$, et g_1, ... , g_m des fonctions μ-récursives totales. Soit f définie de la manière suivante :*

$$f(\vec{n}) = \begin{cases} Si\ \chi_{A_1}(\vec{n}) = 1\ :\ g_1(\vec{n}) \\ ... \\ Si\ \chi_{A_m}(\vec{n}) = 1\ :\ g_m(\vec{n}) \end{cases}$$

Alors f est également μ-récursive et totale.

Démonstration. Rappelons que les fonctions d'addition et de multiplication de deux entiers sont primitives-récursives.

Ensuite, nous remarquons que pour tout $1 \leq i, j \leq m$, $j \neq i$, si $\chi_{A_i} = 1$ alors $\chi_{A_j} = 0$ puisque les ensembles A_1, ... , A_m forment une partition de N^k. Pour cette raison, nous avons :

$$f(\vec{n}) = g_1(\vec{n})\chi_{A_1}(\vec{n}) + ... + g_m(\vec{n})\chi_{A_m}(\vec{n})$$

Cela prouve le lemme. ■

3.2 Vers une preuve des théorèmes de récursion de Kleene

3.2.1 Théorème d'itération

Nous allons commencer par énoncer et prouver le théorème d'itération ou théorème s-m-n qui interviendra dans la preuve des théorèmes de récursion de Kleene.

THÉORÈME 3 (Théorème s-m-n). *Soit $m, n \in \mathbb{N}$ et $\phi_i : \mathbb{N}^{m+n} \to \mathbb{N}$ une fonction μ-récursive. Alors il existe une fonction primitive-récursive S_n^m tel que, pour tout $(x_1, ..., x_{m+n}) \in \mathbb{N}^{m+n}$*
$\phi_{S_n^m(i,x_1,...,x_m)}(x_{m+1}, ..., x_{m+n}) = \phi_i(x_1, ..., x_m, x_{m+1}, ..., x_{m+n})$

Démonstration. Nous allons prouver ce théorème par récurrence sur m.

1. Cas de base, $m = 1$

   On va montrer, étant donné une fonction μ-récursive $\phi_i : \mathbb{N}^{1+n} \to \mathbb{N}$, l'existence d'une fonction primitive-récursive $S_n^1 : \mathbb{N}^2 \to \mathbb{N}$ tel que $\phi_{S_n^1(i,x)}(\vec{y}) = \phi_i(x, \vec{y})$ pour tout $x \in \mathbb{N}$, $\vec{y} \in \mathbb{N}^n$.

   On va définir la fonction S_n^1 comme suit :
   $S_n^1(i, x) = j$ ssi $\phi_j(\vec{y}) = \phi_i(C_x(\vec{y}), P_1^n(\vec{y}), ..., P_n^n(\vec{y}))$
   avec C_x la fonction constante égale à x : $C_x(\vec{y}) = x\ \forall \vec{y} \in \mathbb{N}^n$.

   La fonction ϕ_j s'obtient par composition de la fonction μ-récursive ϕ_i avec des projecteurs et une fonction constante, qui sont primitives-récursives. Ainsi, ϕ_j est μ-récursive, donc il existe une fonction primitive-récursive qui calcule j. Par conséquent, la fonction S_n^1 est bien primitive-récursive.

2. Cas général, $m \geq 1$

   Hypothèse de récurrence : pour tout $n \in \mathbb{N}$, il existe S_n^m primitive-récursive tel que $\phi_{S_n^m(i,x_1,...,x_m)}(x_{m+1},...,x_{m+n}) = \phi_i(x_1,...,x_m,x_{m+1},...,x_{m+n})$

   Montrons que nous pouvons construire S_n^{m+1} à partir de S_{n+1}^m et S_n^1 définie dans l'étape $m = 1$. En effet, soit $\vec{y} \in \mathbb{N}^n$. Nous avons alors :

$$\begin{aligned}\phi_i(x_1,x_2,...,x_m,x_{m+1},\vec{y}) &= \phi_{S_{n+1}^m(i,x_1,...,x_m)}(x_{m+1},\vec{y}) \\ &= \phi_{S_{n+1}^m(i,x_1,...,x_m)}(x_{m+1},\vec{y}) \text{ hypothèse de récurrence} \\ &= \phi_{S_n^1(S_{n+1}^m(i,x_1,...,x_m),x_{m+1})}(\vec{y}) \text{ définition de } S_n^1 \\ &= \phi_{S_n^{m+1}(x_1,...x_m,x_{m+1})}(\vec{y})\end{aligned}$$

   en posant $S_n^{m+1}(x_1,...x_m,x_{m+1}) := S_n^1(S_{n+1}^m(i,x_1,...,x_m),x_{m+1})$
   S_n^{m+1} est alors primitive-récursive en tant que composée de fonctions primitives-récursives.

■

Nous rappelons la signification du théorème s-m-n : il affirme qu'en fixant m paramètres d'une fonction calculable à $n+m$ paramètres, nous obtenons une nouvelle fonction calculable dont le code s'obtient via une fonction primitive-récursive appliquée au code de la fonction de départ et aux valeurs des m paramètres fixés. On appelle cette procédure **évaluation partielle**. Ce théorème a des utilisations au-delà de celle présentée dans le texte.

Quelques remarques sur la preuve, qui peut paraître peu compréhensible au premier abord. Nous avons défini la fonction S_n^1 comme la fonction qui renvoie simplement le code de la fonction ϕ_i en remplaçant sa première variable par une constante. Nous avons ensuite montré que ce code peut se calculer à l'aide d'une fonction primitive-récursive. Enfin, nous avons démontré qu'il est possible de faire cela pour n'importe quel nombre de variables en les remplaçant une par une.

Grâce à tout ce qui a été dit, nous sommes maintenant en mesure de donner les formulations exactes des deux versions du théorème de récursion de Kleene que nous avons vues dans la partie précédente.

3.2.2 Théorèmes de récursion de Kleene

THÉORÈME 4 (Théorème de récursion de Kleene, formulation originale). *Pour toute fonction $f : \mathbb{N}^{1+k} \to \mathbb{N}$ μ-récursive et totale, il existe une fonction μ-récursive $\phi_e : \mathbb{N}^k \to \mathbb{N}$ telle que $\phi_e(\vec{x}) = f(e,\vec{x})$ pour tout $\vec{x} \in \mathbb{N}$.*

Démonstration. On considère la fonction S_k^1 donnée par le théorème s-m-n appliquée à une fonction convenable (on verra quelle est cette fonction au cours de la preuve !).

On définit une fonction g de la manière suivante : $g(y,\vec{x}) := f(S_k^1(y,y),\vec{x})$

La fonction g a été construite par composition à partir de fonctions μ-récursives et totales, par conséquent elle est μ-récursive (et totale) et a un code a, $g = \phi_a$.

En prenant $e = S_k^1(a, a)$, nous avons :

$$\begin{aligned}
\phi_e(\vec{x}) &= \phi_{S_k^1(a,a)}(\vec{x}) \text{ par définition de } e \\
&= \phi_a(a, \vec{x}) \text{ théorème s-m-n appliqué à } \phi_a \\
&= g(a, \vec{x}) \text{ par définition de } \phi_a \\
&= f(S_k^1(a, a), \vec{x}) \text{ par définition de } g \\
&= f(e, \vec{x}) \text{ par définition de } e
\end{aligned}$$

■

Cette preuve, dont les étapes sont quasi-mécaniques, peut paraître contre-intuitive lorsque l'on y réfléchit un peu plus : en effet, g est définie à l'aide de la fonction S_k^1 d'elle-même ! Mais il n'y a rien de contradictoire là-dedans, car si g est μ-récursive, le théorème s-m-n garantit que S_k^1 est primitive-récursive, et si S_k^1 est primitive-récursive, g est μ-récursive par construction.

THÉORÈME 5 (Théorème de point fixe de Rogers). *Pour toute fonction $h : \mathbb{N} \to \mathbb{N}$ μ-récursive et totale, il existe une fonction μ-récursive $\phi_e : \mathbb{N}^k \to \mathbb{N}$ telle que $\phi_e(\vec{x}) = \phi_{h(e)}(\vec{x})$ pour tout $\vec{x} \in \mathbb{N}$.*

Démonstration. On définit une fonction g de la manière suivante :

$$g(t, \vec{x}) = \begin{cases} \text{Si } \phi_t(t) \downarrow : \ \phi_{h(\phi_t(t))}(\vec{x}) \\ \text{Sinon : } g(t, \vec{x}) \uparrow \text{, i.e. } g \text{ n'est pas définie en } (t, \vec{x}) \end{cases}$$

Soit n le code de g, i.e. $g = \phi_n$. Alors, d'après le théorème s-m-n appliqué à g, il existe S_k^1 primitive-récursive telle que

$$g(t, \vec{x}) = \phi_n(t, \vec{x}) = \phi_{S_k^1(n,t)}(\vec{x})$$

Rappelons que n est le code de la fonction g (i.e. ce n'est pas une variable libre). On peut donc considérer la fonction ϕ_m définie de la manière suivante : $\phi_m(t) := S_k^1(n, t)$.
Alors, puisque S_k^1 est primitive-récursive et totale, ϕ_m l'est aussi. En particulier, $\phi_m(m)$ a une valeur bien définie.

Nous avons alors, pour tout $t \in \mathbb{N}$ et $\vec{x} \in \mathbb{N}^k$:

$$\begin{aligned}
\phi_{\phi_m(t)}(\vec{x}) &= \phi_{S_k^1(n,t)}(\vec{x}) \text{ par définition de } \phi_m \\
&= \phi_n(t, \vec{x}) \text{ par définition de } S_k^1 \\
&= g(t, \vec{x}) \text{ par définition, } \phi_n = g \\
&= \phi_{h(\phi_t(t))}(\vec{x}) \text{ si } \phi_t(t) \downarrow
\end{aligned}$$

En particulier, pour $t = m$, nous avons bien $\phi_m(m) \downarrow$ d'après les remarques qui précèdent. Ainsi :

$$\phi_{\phi_m(m)}(\vec{x}) = \phi_{h(\phi_m(m))}(\vec{x})$$

Ainsi, $e := \phi_m(m)$ est un point fixe, ce qui prouve le théorème. ■

3.2.3 Fonctions auto-réplicatrices : les quines

Nous allons reformuler de manière plus rigoureuse ce qui a été dit.

DÉFINITION 10 (Quine). *Une fonction μ-récursive ϕ_e est appelée un* ***quine*** *si $\phi_e(\vec{x}) = e \ \forall \ \vec{x}$.*

L'existence des quines est un corollaire du théorème de récursion de Kleene. En effet, la formulation originale appliquée à la fonction de projection P_1 sur la première variable affirme l'existence d'un e tel que $\phi_e(\vec{x}) = P_1(e, \vec{x}) = e$.
Pour prouver leur existence avec la forme de point fixe de Rogers, il faudrait prendre h tel que $\phi_{h(e)}(\vec{x}) = e$. Autrement dit, $h(e)$ est le code de la fonction constante égale à e, par conséquent h est primitive-récursive d'après le théorème sur le codage.

3.2.4 Théorème de Rice

Dans cette section, nous allons énoncer et prouver le théorème de Rice de manière rigoureuse.

DÉFINITION 11 (Ensemble sémantique). *Un ensemble de codes $A \subset \mathbb{G}$ est appelé* ***ensemble sémantique*** *si, pour tout $x, y \in \mathbb{G}$, le fait que $x \in A$ et $\phi_x \simeq \phi_y$ implique $y \in A$.*

Cette notion formalise l'idée de ne tenir compte que du comportement sémantique d'un programme. En effet, pour un ensemble sémantique A donné, la classe d'équivalence par rapport à $\simeq$ de chaque fonction vérifie l'une des deux choses suivantes : ou bien la totalité de ses éléments sont dans A, ou bien aucun de ses éléments n'est dans A.

THÉORÈME 6 (Théorème de Rice). *Soit $A \subset \mathbb{G}$ un ensemble d'indices. Alors A est décidable si et seulement si $A = \emptyset$ ou $A = \mathbb{G}$.*

Démonstration. Nous allons prouver ce théorème par contradiction. Supposons donc qu'il existe $\emptyset \subsetneq A \subsetneq \mathbb{G}$ qui soit décidable, i.e. χ_A est μ-récursive et totale.

Prenons ensuite $m, n \in \mathbb{N}$ tel que $m \in A$ et $n \notin A$. Définissons une fonction h comme suit :

$$h(x, \vec{y}) = \begin{cases} \text{Si } \chi_A(x) = 0 : \phi_m(\vec{y}) \\ \text{Si } \chi_A(x) = 1 : \phi_n(\vec{y}) \end{cases}$$

Alors h est μ-récursive et totale. Ainsi, d'après la première forme du théorème de récursion de Kleene, il existe $e \in \mathbb{G}$ tel que $h(e, \vec{y}) = \phi_e(\vec{y})$ pour tout $\vec{y}$.

Nous nous intéresserons à la question de l'appartenance de e à l'ensemble A.

- Si $e \in A$, $\chi_A(e) = 1$, par conséquent $\phi_e(\vec{y}) = h(x, \vec{y}) = \phi_n(\vec{y})$. Par conséquent, $\phi_e \simeq \phi_n$. Donc, puisque A est un ensemble sémantique, on devrait avoir $n \in A$, or ce n'est pas le cas. Contradiction.
- Si $e \notin A$, $\chi_A(e) = 0$, par conséquent $\phi_e(\vec{y}) = h(x, \vec{y}) = \phi_m(\vec{y})$. Par conséquent, $\phi_e \simeq \phi_m$. Donc, puisque A est un ensemble sémantique, on devrait avoir $m \notin A$, or ce n'est pas le cas. Contradiction.

Par conséquent, l'hypothèse selon laquelle A était décidable a conduit à une contradiction. Elle est donc fausse. ■

Comme corollaire de ce théorème, il y a notamment le fameux problème de l'arrêt, entre autres.

Corollaire 1 (Problème de l'arrêt). *Soit $\vec{x} \in \mathbb{N}^k$.*
Soit $H = \{n \in \mathbb{G} \ : \ \phi_n(x) \downarrow\}$. Alors H est indécidable.

Corollaire 2 (Problème de la totalité).
Soit $T = \{n \in \mathbb{G} \ : \ \phi_n(x) \downarrow \ \forall \ \vec{x} \in \mathbb{N}^k\}$. Alors T est indécidable.

Corollaire 3 (Problème de l'équivalence). *Soit $m \in \mathbb{N}$.*
Soit $E = \{n \in \mathbb{G} \ : \ \phi_n \simeq \phi_m\}$. Alors E est indécidable.

4 Remerciements et remarques

C'est mon mémoire de licence, encadré par Ekaterina Fokina, qui m'a offert l'occasion d'approfondir mes connaissances de ces résultats magnifiques.

Je remercie mes amis Anaïs Allaire et Quentin Hoenen pour leurs remarques sur la section 2.

Tout au long de ce texte, j'ai tâché de rendre visible les visages derrière les noms mentionnés. Mais je n'ai malheureusement trouvé aucune photo de Henry Gordon Rice (1920-2003).

Références

[1] Quine - rosetta code. http ://rosettacode.org/wiki/Quine.

[2] Bonfante, G., Kaczmarek, M., and Marion, J.-Y. On abstract computer virology from a recursion-theoretic perspective. *Journal in Computer Virology 1*, 3-4 (2006), 45–54.

[3] Madore, D. Quines (self-replicating programs). http ://www.madore.org/ david/computers/quine.html.

[4] Moschovakis, Y. N. Kleene's amazing second recursion theorem. *The Bulletin of Symbolic Logic 16* (2010), 189–239.

[5] Rogers, Jr., H. *Theory of Recursive Functions and Effective Computability*, 3 ed. The MIT Press, 1987.

[6] van Dalen, D. *Logic and Structure*, 4 ed. Springer, 2008.

[7] Weber, R. *Computability Theory*. American Mathematical Society, 2012.