Une fonction d'arrêt constructive !

Introduction

Le théorème d'arrêt nous dit qu'il n'existe pas d'algorithme permettant de calculer l'arrêt des machines de Turing. La différence entre mathématiques classiques et mathématiques constructives est que la première accepte de raisonner sur des concepts de façon extérieure, sur un mode logique utilisant le tiers-exclu face à la totalité des possibles, sans aucunement s'intéresser à une lisibilité des concepts incarné dans la structure dont on parle. C'est typiquement ce que fait la démonstration du théorème d'arrêt quand elle raisonne par l'absurde sur l'arrêt ou non de toutes les machines possibles. De tels raisonnements qui s'appuient sur la logique plutôt que de construire les liens calculatoires effectifs, Je les qualifie de raisonnements en « rupture conceptuelle ». C'est l'utilisation d'une structure mathématique univoque qu'on choisit de plaquer par une « interprétation naturelle » sur une situation mathématique sans pouvoir justifier de la liaison de cette structure ajoutée avec la structure initiale autrement que par l'intuition. C'est ce qu'on fait lorsqu'on dit de chaque machine qu'elle doit produire une réponse arrêt ou non, sans dire comment on l'obtient. C'est ce ce qu'on fait de façon générale, quand on suppose que l'ensemble des propositions mathématiques possède une valeur de vérité Vraie ou Fausse, sans expliquer comment on obtient cette valeur. Il suffit de supposer l'existence d'une valeur de vérité et de manipuler la structure booléenne de la vérité sur l'ensemble des propositions, et l'on obtient bon nombre conclusion. Le tiers exclu consiste à accepter de dire que de toute proposition A qu'elle est vraie ou fausse, on en déduit que Non(non(A))=A sans jamais avoir cherché à entrer dans les raisons qui produisent A ou non A. Remarquons que le théorème de Gödel construit une phrase dont la « vérité » ou la « fausseté » ne peut pas se démontrer. Ce qui invite à penser à une voie tierce.

Dans un cadre purement algorithmique, je propose de construire un raisonnement en rupture conceptuelle qui permet de transgresser l'impossibilité du théorème d'arrêt. Je propose donc, malgré l'interdiction posé par le théorème, de construire un algorithme d'arrêt purement constructif dans le cadre des machines de Turing. On étudiera bien sûr le rapport qui existe entre la contradiction et cet algorithme.

La notion d'arrêt est un concept humainement naturel. Un algorithme tourne une étape après l'autre, dans un rythme univoque. Plongées dans une machine de Turing, les étapes présentent une apparence limpide d'univocité. Il semble donc évident de pouvoir statuer sur le fait qu'un algorithme s'arrête ou non. Seulement le concept « d'arrêt/non-arrêt » est un concept en rupture, il n'existe pas directement dans le programme de la machine de Turing. Pour qu'un calcul produire la réponse « s'arrête ou non », il faudra une « interprétation ». Plus précisément, on code souvent l'arrêt par un état qui signifie « arrêt de la machine » qui est donc une notion effective d'arrêt. Mais c'est insuffisant, car rien n'est fait pour le non-arrêt qui ne possède aucune notion effective. Comment peut-on lier un constat effectif au concept de « non-arrêt » ? Ce n'est pas complètement impossible. On peut par exemple contrôler l'existence d'une boucle qui se répète infiniment, il faudra alors lier un processus qui cherche les boucles infinies à un processus qui décrète le non-arrêt. Mais ce n'est pas suffisant. Comment faire pour un comportement complexe dont on ne sait pas s'il a une fin, comment avoir une marque systématique de non-arrêt ? Si une boucle peut être identifiée, comment reconnaître parmi les comportements non-linéaires complexes ceux qui « cause » un non-arrêt. Il reste le moyen « d'attendre à l'infini », mais cela a-t-il du sens ?

Un des avantages des mathématiques sur le calcul est justement de pouvoir user de concepts en rupture sur l'effectivité. L'arrêt est facile à prouver, car il suffit d'exécuter le programme et d'attendre. Si on sait qu'il s'arrête, ça finira par arriver, et on aura la preuve de l'arrêt. En fait, c'est typiquement le schéma de preuve qui satisfait intellectuellement : l'idée d'une répétition qui prend fin indépendamment du temps d'attente. Le problème survient lorsqu'on ne sait pas si un algorithme s'arrête, on ne peut s'engager dans cette voie car si l'algorithme ne s'arrête pas, on n'atteindra jamais de preuve, il faut changer de registre et faire une preuve par analyse du code source. Seulement la complexité d'une telle analyse est illimitée, car presque toutes les preuves d'existence mathématique reviennent à une preuve d'arrêt quand on test par force brute. On est

alors tenté de croire qu'aucune analyse générale ne peut statuer sur un non-arrêt. Dire d'un programme qu'il s'arrête ou non est alors un discours idéal « situé depuis l'extérieur ». Quelle pertinence possède ce concept ?
En fait, avec un peu d'imagination, on peut aussi construire des concepts en rupture qui permettent de « savoir » si une fonction ne s'arrête pas. Pour cela, je propose de définir le concept d'enveloppe maximale d'une machine de Turing.

Enveloppe maximale d'une machine de Turing.

Une machine de Turing Universelle peut simuler toute machine de Turing en codant sa fonction de transition sur le ruban selon un mécanisme plutôt simple : à chaque fois qu'elle aura besoin d'une information, elle viendra la cherchée là où on l'a écrite. De plus l'ensemble de toutes les machines de Turing est simple à lister : il suffit de construire tous les mots finis dans l'ordre lexicographique pour chaque taille donnée à partir de l'alphabet accepté par une machine. Il faut ensuite vérifier parmi les mots ceux qui forment une machine. Pour cela, il faut choisir un codage qui désigne le nombre d'états et les actions à produire pour chaque état et chaque lettre lue. Il n'y a donc aucune ambiguïté à vérifier les nombres qui sont des machines et ceux qui n'en sont pas.
On suppose donc une machine unique qui va exécuter en parallèle toutes les machines, nommons m la variable qui désigne les différentes machines de Turing parcourues par le programme général. Remarquons que le choix du type de machine de Turing possède peu d'importance (taille de l'alphabet, méthode d'arrêt, ruban d'entrée et de sortie ou non,…).

Définition de la fonction q (quantité d'étapes) :
Les concepts qui suivent ne sont pas constructifs, ils sont définis par un regard d'évidence extérieure, sans être définis par un calcul. On procédera à une traduction calculatoire ultérieurement.
D'un point de vu conceptuel, on peut considérer qu'il existe des machines qui s'arrêtent et d'autres qui ne s'arrêtent pas. Dès lors qu'on pense qu'il existe une telle catégorisation, indépendamment de la possibilité de trouver la réponse, j'appelle cela « accepter le concept d'arrêt ». De même, on peut aussi supposer que chaque machine qui s'arrête parcourt un certain nombre d'étapes avant son arrêt, quoi de moins « naturel ». Sous cette hypothèse, on peut alors raisonner en rupture conceptuelle : pour chaque machine de Turing m, soit elle s'arrête et il existe un nombre d'étapes $q(m)$ jusqu'à l'arrêt, soit elle ne s'arrête pas et on pose $q(n)=0$.

Définition de la fonction enveloppe maximale M
Comme la fonction q est définie pour tout n, il existe parmi les n premières machines un maximum pour la fonction q. C'est une fonction croissante en n, définie par $M(n) = max\{q(m), 1 \leq m \leq n\}$.

Définition de la suite s_n des enveloppes maximales
$M(n)$ est croissante, mais souvent constante. On définie alors la suite s_n par les valeurs de n où $M(n)$ croît : $s_1=1$ et $s_{n+1} = min\{m ; m>s_n \text{ et } M(m)>M(s_n)\}$. C'est une suite de valeurs localement maximales.

Définition du non-arrêt
Par définition, la Machine de Turing numéro n ne s'arrête pas si le nombre d'étapes effectuées par la machine a dépassé $M(n)$.

Ces concepts non calculatoires permettent de raisonner sur les machines de Turing univoquement (mais en rupture). Il est alors possible d'implémenter ces fonctions dans une machine de Turing de façon constructive, il est donc possible de « calculer » le non-arrêt.
Bien sûr, on ne va les définir ici que d'un point de vu algorithmique de haut-niveau, une implémentation en direct sur une machine de Turing serait excessivement complexe sans disposer d'un compilateur automatique. Ainsi en principe, rien n'empêche une machine de Turing de recevoir les calculs qui vont être décrits.

Fonction d'arrêt constructive

Plaçons-nous dans le cadre des machines de Turing avec l'alphabet d'écriture {0,1}, l'arrêt d'une machine étant provoqué par un déplacement à gauche sur la case initiale qui est le seul mouvement impossible (car le ruban part vers la droite).
Il est possible de construire une machine de Turing Universelle qui exécute en parallèle sur un unique ruban l'ensemble de toutes les machines. Il suffit de placer toutes les cases de chaque machine sur l'unique ruban par la formule : (machine i, case j) sur la case [j(j-1)+2ji+(i-1)(i-2)]/2 (on prend successivement chaque diagonale). Il faudra y ajouter une infinité de place technique pour stocker des données en mémoire et permettre de se déplacer sur le ruban en suivant des stratégies variables. Une fois les cases du ruban attribuées, il faudra exécuter tour à tour les étapes de chaque machine (selon une formule d'exécution similaire, par diagonales).
Voici à grands traits le programme de la machine de Turing universelle:

La boucle principale itère sur la variable D (pour diagonale) :
1-Ecrire la machine de Turing n°D (qui est le numéro D) sur les cases dévolues. Analyser si ce n° D est une machine de Turing selon le format prédéfini (nombre d'états, puis écriture, déplacement et nouvel état pour chaque état). Initialiser les variables techniques qui vont permettre de suivre son déroulement
2-Exécuter une étape de chaque machine de 1 à D. Conserver le nombre d'étapes, la position de la tête, le numéro d'état de chaque machine.
3-Surveillance et action lors du déroulement général (variables spécifiques).

La simulation effectuera les étapes de chaque machine en diagonale : quand n étapes seront réalisées pour la machine 1, on aura n-1 pour la machine 2, … ,1 étape pour la machine n (si c'est une machine valide), jusqu'à ce que la nouvelle diagonale commence… Dans la partie 3 de la gestion multi-tâche, on procèdera en particulier à la construction des enveloppes maximales : le nombre d'étapes effectué est mis en mémoire pour chaque machine et dès qu'une machine s'arrête (déplacement à gauche sur la case 1) on teste si c'est une enveloppe maximale provisoire. En effet, chaque nouvel arrêt supérieur à tout précédent se propose à la candidature d'une enveloppe maximale définitive, on le nomme « enveloppe maximale provisoire », parce qu'on n'est jamais à l'abri qu'un algorithme numéro k non encore arrêté de rang inférieur s'arrête et détruise les enveloppes provisoires supérieures à k. La procédure est simple : pour chaque machine qui s'arrête, on vérifie si le nombre d'étapes effectué est plus grand que l'enveloppe maximale provisoire. Si c'est le cas, on inscrit la nouvelle valeur d'enveloppe maximale provisoire sur toutes les machines qui suivent. On va aussi marquer les non-arrêts potentiels : à chaque fois qu'une machine produit un nombre d'étapes supérieur à son enveloppe maximale provisoire, on peut marquer la machine comme « ne s'arrêtant provisoirement pas ». Bien sûr certaines d'entre elles pourront s'arrêter et devenir le lieu d'une enveloppe maximale.
A chaque machine de Turing *n,* on va attribuer une valeur de sortie sur une place technique du ruban. Il y aura 4 valeurs de sortie possibles:
0 pour « pas encore de réponse »: cette valeur est inscrite au moment de la mise en route de la simulation d'une machine. On l'inscrira aussi à chaque fois qu'une nouvelle enveloppe maximale émerge et dépasse le nombre d'étapes effectué par la machine *n*.
1 pour «machine arrêté »: on l'inscrit dès qu'une machine s'arrête.
2 pour « supposition de non-arrêt » : on l'inscrit dès qu'une machine dépasse son enveloppe maximale provisoire.
3 pour «non-machine» : on l'inscrit après création lors du test de format si le nombre n'est pas une machine de Turing.
Ce nombre est stocké pour chaque machine depuis l'origine de sa simulation. On note $H_p(n)$ cette fonction d'arrêt provisoire inscrite pour la n-ième machine après p étapes de la machine universelle.

Lecture de la fonction d'arrêt

Le raisonnement central est le suivant : comme pour tout *n*, il « existe » un maximum *M(n)* d'étapes après quoi, il n'y a plus d'arrêt possible, on peut affirmer que les machines qui ont franchit ce cap ne s'arrêteront plus. Dès lors que *M(n)+1* étapes seront effectuées sur une machine *m* (avec *m<n)* sans s'être arrêtée, elle ne s'arrêtera plus.

La vraie difficulté est que *M(n)* n'est pas calculable. Mais ce n'est pas rédhibitoire en « logique ». Car en raisonnant avec ces concepts « naturels », on peut affirmer qu'en allant assez loin dans le processus, les valeurs provisoires d'enveloppe maximale seront forcément atteintes et donc que les valeurs d'enveloppe maximale provisoire deviendront définitives à partir d'un certain rang. C'est l'essence même de la définition de ces enveloppes. La valeur de sortie de chaque machine *m* va donc « converger » pour tout *n* en un temps fini (mais inconnu). Ainsi, à l'aide de ce raisonnement qui consiste à comparer les étapes aux valeurs d'enveloppe maximale provisoire, la convergence aura lieu pour les machines qui s'arrêtent comme pour celles qui ne s'arrêtent pas.

On est face au problème de la définition en mathématique, et en particulier au problème de la définition en rupture conceptuelle. Même si on sait que ce nombre d'étape « existe » (mot qui reste à définir), on est incapable de dire quand cela a lieu. On est toujours dans l'attente qu'une autre machine, en dessous, s'arrête ultérieurement. Mais on « sait », selon un raisonnement logique en rupture, qu'il y a un temps pour chaque *n*, où cela sera terminé. Donc il y a bien un maximum et donc une disctintion entre arrêt et non-arrêt. Les concepts en rupture permettent de parler des choses sans mettre les mains dans la technique pour rechercher sil y a une liaison de sens avec la structure décrite. Le véritable enjeu consisterait à savoir quand a lieu l'arrêt…

Je crois que c'est dans un tel lieu qu'apparait la nuance entre le « calcul » et les « mathématiques ». Ces dernières possèdent la liberté conceptuelle : on peut s'affirmer que pour tout n, en allant assez loin en étape le ruban est stabilisé pour toutes les cases avant *n* et qu'on obtien donc une réponse. Même si tout cela n'est là qu'une vue de l'esprit. On peut donc écrire la fonction d'arrêt sous un mode conforme aux mathématiques classiques, : $\forall n \exists p \forall t , t>p \Rightarrow H_t(n)=H_p(n)$. $H_p(n)$ qu'on peut écrire H(n) est la valeur d'arrêt. En réalité, on a même mieux, une convergence par tronçon : $\forall n \exists p \forall m \forall t , m\leq n$ ET $t>p \Rightarrow H_t(m)=H_p(m)$

On a satisfait au standard analytique des mathématiques classiques par un procédé purement constructif (pour le processus de fabrication, pas pour l'interprétation en lecture).

Et la diagonale inverse ?

Etant donné qu'on a construit la fonction d'arrêt complète, qu'en est-il de la contradiction habituelle qui fait valoir l'impossibilité de la fonction d'arrêt ? En effet, dans le théorème d'arrêt, on a construit la diagonale inverse à partir d'une structure fonctionnelle avec une variable d'entrée. Mais on a besoin d'un cadre fonctionnel pour réfléchir à la question que pose le théorème d'arrêt classique. C'est une « fonction» qui renvoie une réponse. Ce qu'on vient de faire avec les machines de Turing élémentaires, on peut facilement le transposer aux machines de Turing « fonctionnelle », c'est-à-dire avec un ruban qui prend une valeur en entrée (qui peut aussi représenter un programme). Au total, la description change assez peu.

Modifications structurelles

Les machines de Turing fonctionnelle acceptent une valeur d'entrée (qu'on met sur un ruban séparé ou sur le ruban initial). Toutes les machines valides devront recopier l'entrée finie fournie à la machine universelle sur « leur propre ruban » avant de commencer l'exécution. Il faut donc aussi modifier la machine universelle qui simule tout cela en machine fonctionnelle ayant un ruban d'entrée. Le calcul complet reste identique pour toutes les machines. La machine universelle teste quand une étape est effectuée sur la machine désignée par la valeur donnée en entrée.

Dans un cas d'écriture de la valeur diagonale (la valeur d'entrée égale au numéro de la machine simulée), on recopiera le résultat sur le ruban de sortie principal. Celui-ci est réinscriptible, il recevra donc les valeurs d'arrêt provisoire de la machine diagonale (parmi les 4 valeurs de sortie possible).

Dans ce cadre fonctionnel, on peut simplifier le programme : au lieu de faire un suivi des enveloppes maximales pour chaque machine et renvoyer une réponse pour chacune, on se concentrera de suivre la machine donnée en entrée du programme principal.
Les difficultés passées sous silence concernent la gestion de la bande mémoire (l'essentiel étant traité dans le concept de machine universelle) et le codage de chacun des concepts cités. Malgré la technicité de l'entreprise, je ne pense pas que ce soit un motif de rejet du raisonnement général.

Valeur contradictoire ?
Jusqu'ici, on a construit une machine qui extrait la « valeur diagonale », il faut la modifier une dernière fois en construisant l'inverse pour aboutir à la contradiction. Pour on ajoute un morceau de programme qui produit l'inversion: « si la valeur de sortie est 'arrêt', on lance une boucle, si c'est 'non-arrêt', on s'arrête. » La contradiction doit avoir lieu quand on fournit le numéro de cette même fonction en entrée du programme principal. Et là, les difficultés interprétatives commencent. Avec la question cruciale : comment pourrait-on faire une action à partir d'une valeur d'arrêt étant donné qu'on ignore quand cette valeur est atteinte ? On peut penser que le programme ne se terminera jamais parce que le test ne dira jamais définitivement « ne se termine pas ». La convergence est théorique.

Pas de contradiction sans inversion
Si l'on n'implémente pas le morceau de programme qui « inverse la réponse », la réponse est facile. Comme on connait le déroulement du programme, on peut déterminer la valeur de sortie. En effet, le programme n'est pas fait pour s'arrêter, lorsque le nombre d'étapes effectué dépassera le nombre d'étapes d'une enveloppe maximale provisoire elle passera à 2 :non. (Selon le mode choisi, elle pourra éventuellement repasser ou non à 0 si une valeur de terminaison plus grande se manifeste). Dès que l'enveloppe maximale définitive sera atteinte, la réponse sera définitivement stabilisée à 2. On peut ainsi garantir conceptuellement qu'au bout d'un temps fini, on aura la valeur « non-arrêt ». Remarquons que cette valeur d'arrêt n'est pas issue d'un calcul logique à partir d'une valeur trouvée à l'intérieur du programme dans une strate inférieure qui simule le programme. En testant si le programme universel d'arrêt s'arrête, la réponse sera tout simplement 'non'. Elle sera stabilisée au bout d'un long temps de calcul (sans savoir précisément quand).
L'existence de strates inférieures qui possèdent, elles aussi, les mêmes types de processus et le même résultat, ne pose aucun problème, y compris l'idée d'une mise en miroir infinie. Chacune des valeurs de chaque strate est stockée en des lieux différents et aucune lecture de ces valeurs n'agit dans la modification d'une des valeurs des strates supérieures.

Question de limite ?
On comprend ainsi que la contradiction est portée par l'encapsulation dans ce petit programme d'inversion « si la réponse est 'arrêt' alors poursuivre indéfiniment, sinon s'arrêter ». D'où vient le problème ? Est-il un problème de limite ? On ne sait pas quand la valeur est acquise, on peut se dire que la réponse s'acquiert dans un processus limite. Quand la réponse n'existe que sous forme d'un processus « limite » infini (comme pour la définition d'un réel), cela exclue de pouvoir constructivement lire la réponse et réagir en un temps constructif fini. Cela n'est pas possible sans rupture conceptuelle, qui elle seule, peut considérer la réponse comme acquise. D'où l'incompatibilité de la dimension fonctionnelle : on ne peut pas disposer de la réponse dans un calcul. On peut « savoir » qu'il existe une valeur de sortie, mais « pas suffisamment » pour la faire entrer dans un calcul constructif. C'est le moment d'exprimer le parallèle qu'on peut faire avec la définition d'un réel par ses décimales, est-il suffisamment défini pour dire qu'on le connait ? Un réel quelconque pose d'ailleurs un problème plus grand qu'ici : contrairement à l'arrêt, on ne peut pas dire qu'on le « connait » au bout d'un temps fini, car il faudrait avoir choisi *toutes* ses décimales pour le « connaître ».
C'est pourquoi, on possède ici bien mieux qu'un processus limite. On peut affirmer qu'il existe une valeur finie où le résultat est atteint. Le problème est qu'on ne sait pas quand. On pourrait essayer de l'estimer, de la majorer et dire : « Quand cette valeur est dépassée, lire le résultat et appliquer la valeur d'inversion ». Mais cela pose un autre problème. Pour majorer constructivement une valeur qui est « maximale » pour une taille donnée, il faudrait intégrer cette valeur maximale dans le programme, or par définition, il n'existe aucun programme en dessous de cette taille pour

l'atteindre, elle ne peut être qu'à l'intérieur d'un programme plus grand et donc pas dans notre programme. Pour coder cette valeur de majoration dans le programme, la taille du code source va augmenter, ce qui demande une nouvelle valeur maximale pour majorer, et ainsi de suite. Il n'est donc pas possible de majorer une telle valeur à l'intérieur du programme. Cela en dit long sur la nature de la complexité.

Inaccessibilité des enveloppes maximales

Si effectivement, il y a fort à penser que les enveloppes maximales sont d'une complexité irréductible qui ne pourra pas être rendue constructivement avec des concepts simples, la singularité de chaque machine de Turing me fait penser de façon analogique à la singularité de chaque nombre entier qui rend difficile le positionnement anticipatif des nombres premiers. Mais c'est pire ici, car il n'y a pas de régularité dans la succession des diviseurs, il s'agit d'une interprétation complexe et singulière à déployer pour chaque machine. On ne maîtrise pas cette complexité parce qu'…elle est complexe. La construction artificielle de la fonction d'arrêt prend le problème à l'envers : pour pouvoir dire qu'un numéro est une enveloppe maximale, il faudrait pouvoir montrer qu'elle s'arrête bien et que toutes celles qui sont avant ne s'arrêtent pas sur une valeur supérieure ainsi que la preuve de toutes celles qui ne s'arrêtent pas. Il faudrait « tout savoir sur les valeurs d'avant ». Mettre les mains dans le moteur consiste à maîtriser une somme de « vraies » complexités. Pour poursuivre l'analogie avec les nombres premiers, les deux sont conçues par des informations en creux : « il n'y a aucun diviseurs présent à cette position donc c'est un nombre premier », « il n'y a personne qui s'arrête avec plus d'étapes avant donc c'est une enveloppe maximale ». Cette définition négative demande toute la connaissance qui les entoure pour les maîtriser, c'est ce qui les rend si difficiles d'accès.

Arrêt effectif ou codage

Un programme manipule par codage d'informations. Ainsi l'arrêt peut-être soit codé, soit effectif. Quand on simule un mécanisme, on le code et c'est une interprétation qui apporte la réponse à partir de l'information codée. En effet, si on choisi un arrêt effectif pour la machine, où qu'il soit, on ne pourra plus calculer quoi que ce soit une fois que le programme est arrêté et c'est regrettable pour notre fonction d'arrêt qui est infini par construction. L'intérêt du codage interprétatif est de pouvoir donner une réponse sans produire d'arrêt. Le codage et l'interprétation permettent une grande souplesse d'action, ils offrent beaucoup de possibilités pour implémenter la « fonction inverse » qui doit produire la contradiction.

Pour aller plus loin, on comprend bien qu'en écrivant 2 pour « arrêt » sur la bande de sortie, cela possède un sens subjectif. Cela ne dit pas que la fonction va s'arrêter (c'est codé au lieu d'être effectif), mais cela définit la valeur de sortie de la fonction. Or n'est-ce pas ce qu'on veut lorsqu'on dit vouloir calculer l'arrêt ? En mathématique tout est histoire d'interprétation. Le support de l'interprétation peut-il être imposé ? A priori, on est libre sur le choix du support qui n'a pas être imposé. Ainsi « l'arrêt » n'impose pas nécessairement l'arrêt effectif de la machine. C'est notre esprit qui attend de façon plus ou moins hégémonique de donner un sens maximal à ce qu'on dit, à ce qu'on veut. Si un lien univoque constructif a été créé entre le calcul et une valeur dont on connait la convergence, c'est suffisant en terme mathématique. On peut donc dire qu'au sens des mathématiques classique, la fonction que nous avons construite nous donne bien notre réponse. On voulait un résultat fonctionnel, or arrêter un algorithme n'est pas spécifiquement un résultat fonctionnel, de plus c'est incompatible avec les calculs nécessaires. Donc une interprétation fonctionnelle et non « actionnelle » semble indispensable. On remarquera que toute contradiction a alors disparu. Mais la formulation devient un peu gênante : « si le programme ne s'arrête pas arrêter la machine. ». Dit ainsi, on n'attend pas une valeur de retour codée. Face à une phrase comme « si le programme ne s'arrête pas renvoyer la valeur arrêt et arrêter la machine », on se devrait de retirer le morceau de phrase « arrêter la machine » qu'on ne peut pas mettre en œuvre pour le besoin des calculs. C'est donc face aux machines dont on peut programmer l'arrêt qu'on aura impossibilité de donner une fonction d'arrêt. Si la machine universelle qui simule devait s'arrêter, il n'y aurait pas de résultat. Mais à l'inverse, on peut juste simuler. Il m'apparait donc plusieurs alternatives :

-exécuter l'arrêt à une réponse provisoire sans garanti d'être définitif, la réponse n'a pas de garantie de validité.

-Transformer l'application pratique de l'arrêt en un calcul codé interprétable. Car simuler, c'est nécessairement interpréter. On se concentrera alors sur des calculs à partir de 0 et 1 qui sont manipulables avec beaucoup plus de souplesse que 'arrêt' ou 'non-arrêt' effectif.
- on peut ajouter le « programme creux » dont on reparlera après.

Maintenant, on ne peut pas jouer impunément sur les interprétations. En faisant varier le sens sous de multiples concepts, certains sens sont plus contraignants que d'autres. En restreignant le contexte à une interprétation constructive finie qui renvoie une réponse fonctionnelle à partir du résultat sur lui-même, on arrive nécessairement à une contradiction.
Calculer l'arrêt d'une fonction qui fait référence à elle-même dans une boucle sans fin tout en devant s'arrêter est contradictoire. Mais la contradiction n'est pas due à l'énoncé de l'arrêt : comme toute machine de Turing est finie, on pourrait espérer en lisant la fonction lire sa finitude ou non par la connaissance de la de la complexité sous-jacente des algorithmes. Avec cette procédure d'inversion, on n'a pas mis le nez dans le moteur. Il s'agit d'une lecture de nature interprétative, extérieure, en rupture. En programmant la question de façon précise, il y a beaucoup d'interprétation à trancher, (comme dans le cas du paradoxe de Richard). Il est clair qu'en plongeant extérieurement le concept d'arrêt dans une structure binaire, il y a facilement contradiction en la demeure. Vu au niveau de la fonction diagonale, il s'agit d'une contradiction de définition.
Maintenant, en élargissant le concept de « réponse », la fonction diagonale inverse n'est pas incompatible avec certains types de réponse. Cette contradiction n'est donc pas suffisante pour rejeter toute fonction d'arrêt. L'approche la plus « raisonnable » consisterait à produire une analyse de la complexité de chaque programme pour y trouver des raisons structurelles d'arrêt ou non. C'est là qu'on se trouve confronté et très vite arrêté par… la complexité.

Raisonnement à priori
On peut procéder de façon radicalement différente : la valeur de réponse est connue, puisqu'il faut une infinité d'étapes pour être certain de converger, la valeur de réponse est « non-arrêt ». Il n'y a aucune hésitation à partir de la conception de notre programme. Puisqu'il faut produire l'inverse, on peut donc décider de dire au programme d'inversion « arrêt direct». Et c'est fini avant même de commencer.
On est dans une situation comparable au paradoxe de Berry : « Soit n le plus petit nombre qui n'est pas définissable en moins de dix-huit mots » ou pire : « soit x ce nombre défini en 8 mots ». Dans ces deux cas, la seule contrainte porte sur la définition elle-même, on ne dresse aucune contrainte sur le nombre. Ce qui laisse toute liberté pour produire contradiction (ou tautologie) par auto-référence sur la définition, indépendamment du nombre. On se retrouve dans la même situation pour donner une réponse à « l'inverse de l'arrêt » appliqué à « l'inverse de l'arrêt ». Pourquoi alors ne pas réaliser l'opération de réponse de façon arbitraire et finie. Puisque le programme principal doit donner 'non-arrêt', on décide de rendre 'arrêt'. Aucun <calcul de l'arrêt> ne sera jamais effectué, la réponse provient d'une analyse réalisée a priori. La réponse devient purement extérieure et subjective, non fondée sur un « calcul ». Or une telle pratique change complètement le sens donné à « si la valeur calculée est 'arrêt' alors… ». Mais en changeant la nature des « calculs », on change aussi la nature de ce qu'on observe sur les calculs, faut-il maintenir la validité de ce lien interprétatif ?
En résumé, pour ne plus se voir reprocher de donner une réponse en un temps inconnu, comme on connait la réponse à l'avance, on peut donner la réponse dans une case spécifique inscriptible une seule fois et puis continuer les calculs... On n'a plus de contradiction, mais la réponse n'est pas le « résultat d'un calcul>. Et il y a rupture de la cohérence entre la réponse et l'observation de ce qui est fait.

Un programme creux
Côté arrêt effectif, si on ne fait que « coder l'arrêt » mais qu'il n'est jamais effectué, on peut faire disparaitre la contradiction. En effet, il est possible d'implémenter le « programme d'inversion » de la façon suivante : on attribue une 5ième valeur de sortie qu'on appelle « non-arrêt définitif » et l'on teste à chaque fois que l'on vient écrire dans la case de sortie si la valeur inscrite est 5. Si c'est le cas, on peut carrément stopper le programme (ou codé l'arrêt). Puisqu'aucune partie du code ne

viendra jamais écrire ce nombre cinq (ou bien si une autre partie qui explique quand l'écrire ne s'exécute jamais non plus), ce code d'arrêt n'aura aucun effet. Rien n'empêche d'écrire un morceau de programme qui ne s'exécutera jamais. Dans ce cas, le programme d'inversion est tout de même été réellement été implémenter. Et le programme ne s'arrêtera jamais soit parce que le module arrêt ne s'arrête pas soit parce qu'il s'arrête et que la boucle qui suit lui dit de ne pas s'arrêter.
Cela donne à réfléchir : écrire ce programme inverse qu'il ne s'exécutera pas ne conduit plus à une contradiction. Par contre le module arrêt ne fournit pas toujours une « réponse » dans un sens fini du terme. La contradiction vient de la nécessité d'obtenir une réponse finie par la fonction arrêt.

Le contraire de soi
En lisant le programme « inverse » avec toute les « attentes naturelles », ce programme possède une définition contradictoire : La fonction d'arrêt devant inévitablement s'arrêté, la fonction devrait systématiquement répondre « non arrêt ». Mais on attend aussi que la fonction donne la réponse inverse d'elle-même. Impossible de se définir comme étant différent de soi. Par contre, c'est possible si on fait intervenir « plusieurs occurrences différentes de soi » dans un processus alternée infini qui offre une lecture dynamique de l'identification : à chaque strate, on donne une réponse différente. Par contre, il ne faut pas le faire par simple lecture du contraire de la strate inférieure, sinon le processus est infini et ne possède pas d'initialisation. Il n'y aurait donc aucune « réponse ». On peut donc produire une réponse arbitrairement alternée à chaque strate, ce qui est une nouvelle façon de lever l'auto-contradiction.

Suivi au fil du calcul
Puisqu'on peut coder l'arrêt, on codera la réponse plutôt que de l'exécuter. Comme le suggère notre fonction d'arrêt, on peut produire une valeur qui évolue au cours du calcul : on écrit 1 (pour « arrêt ») ou 2 (pour « non-arrêt »), sauf qu'on écrit la valeur contraire à celle obtenue par la fonction. La réponse définitive s'obtiendra à partir d'un certain rang inconnu et cette réponse sera « arrêt ». Eventuellement, selon la programmation, la valeur repassera par plusieurs valeurs, mais elle convergera de façons certaines vers cette valeur). Il n'y a plus de contradiction lorsque la réponse est donnée par « une écriture » au lieu de « l'action » de l'arrêt. La réponse converge vers une valeur unique et ne dépend pas des strates inférieures.

Maintenant pour satisfaire à la dépendance des réponses envers les strates inférieures, on peut agrémenter notre algorithme et recopier toutes réponses calculées des strates inférieures vers les strates supérieures en les inversant. On peut d'ailleurs implémenter cette recopie de façon très variée. Une recopie systématique et inversée produira une « danse folklorique et incessante des réponses » qui laisse bien loin l'idée de réponse unique. Ce qui constitue une autre interprétation possible de la fonction d'inversion, certes pas la plus naturelle puisqu'on a abandonné l'idée d'unicité des identités et des valeurs de la réponse au profit d'un calcul de dépendance.

Le problème central : les deux sources de la réponse
Après avoir abondamment joué avec les interprétations, tout s'éclaire quand on comprend que la question posée possède une ambiguïté due à l'existence de deux sources différentes pour produire la réponse. En remarquant que ces sources ne peuvent pas s'accorder sans contradiction. En effet, quand on demande l'arrêt de la fonction inverse appliquée à elle-même, on attend que :
- La réponse soit donnée par la lecture de la réponse dans la strate inférieure à qui on appliquera l'inverse.
- La réponse soit donnée par un calcul d'arrêt global à partir du code source pour qui on appliquera l'inverse.

Or ces deux sources correspondent à deux recherches algorithmiques et conceptuelles très différentes produisant des réponses différentes. Il faut choisir entre les deux sources. Notre algorithme est typiquement un exemple de réponse qui proviendrait d'un calcul sans considération pour la réponse du même programme dans la strate inférieure. « La réponse de non-arrêt » provient de la comparaison des étapes avec les enveloppes maximales réalisées à partir des autres programmes. « La réponse de l'arrêt », elle, provient de l'analyse du mouvement de la tête de lecture (dans le cas où l'on considère que l'arrêt est provoqué par un déplacement à gauche sur la case initiale). A aucun moment, la réponse ne cherche dans le sous-programme de la strate

inférieur. On est bien en train d'essayer de donner un sens pratique à « s'arrête » et « ne s'arrête pas » sans interprétation de conformité avec la strate en dessous.
Or la contradiction porte sur l'idée que la réponse donnée doit être l'inverse de la strate inférieure. Dans notre algorithme, toutes les strates auront la même réponse (au bout d'un temps qui s'allonge dans chaque strate) qui sera « contraire de non-arrêt » soit « arrêt ». Cette réponse est trouvée par un calcul sur l'exécution qui ne dépend pas de la strate inférieure. Le raisonnement qui produit la contradiction supposait l'existence de ce lien. Il y a confusion entre la réponse de arret(inverse(inverse)) dont la réponse est « non-arret » et l'arrêt potentiel de ce programme pour donner la réponse. Pour observer la disparition de la contradiction, le mieux sera de la décrire à la fin du prolongement vertigineux ci-dessous.

Prolongement vertigineux
Maintenant qu'on sait que notre algorithme ne produit pas de contradiction par l'inverse, il ne reste plus que le problème de l'ignorance du moment de l'arrêt pour obtenir une fonction d'arrêt complètement constructive. Je propose alors une tentative audacieuse : il suffit de lire la réponse assez tardivement pour espérer disposer de la réponse de façon effective. Je propose donc de coder un arrêt pour lire la réponse donnée : pour une machine de Turing m qui effectue l'étape numéro q, au lieu de d'écrire une réponse provisoire dès que q>M(m), je propose d'écrire une unique réponse dès que q>E'_m . Pour définir E'_m, on commence par définir E_m « l'enveloppe itérée » par E_1=M(n) et $E_{n+1}=M(\sum_{k=1}^{E_n} q(k))$. Ensuite E'_m consiste à la même définition mais pour les enveloppes provisoires. Sauf que pendant le processus, de nombreuses k-ième enveloppes provisoires vont se transformer. E'_m correspondra à un processus qui permettrait de construire toute la chaine des E_i jusqu'à E_m sans qu'il n'y ait plus de transformation jusqu'à E_m, toutes les enveloppes ont été stabilisées (tant que M(n) change, on reprend tout depuis le début, et tant qu'apparait une nouvelle enveloppe maximale inférieur, on reprend tout depuis cette nouvelle valeur). Tout ce processus est complètement constructif puisqu'il repose sur des enveloppes maximales provisoires, il attend juste d'être stabilisé. C'est la confiance en l'existence d'une stabilisation qui garantie une finalité au processus. Puisqu'il fallait aller très loin pour stabiliser le processus, quoi de plus grand que de prendre appui sur les enveloppes maximales. Stabiliser une enveloppe maximale « très » lointaine semble un candidat idéal. Puisque les enveloppes maximales correspondent à des programmes ayant un nombre d'étapes maximal, on peut supposer que les En sont effectivement de très grand nombres. Puisque chacun prend appui sur chaque enveloppe maximale précédente, on peut envisager une croissance sans commune mesure. Le programme n'est pas très long à écrire, il en est autrement du nombre d'étapes qu'il doit faire.
Ce processus table sur la confiance dans la croissance étalée des Em : à chaque étape, il faut attendre qu'apparaisse le Ei suivant, ce qui oblige à aller suffisamment loin, ce qui oblige l'apparition de la stabilisation des Ei intermédiaires provisoires. C'est un processus à la fois auto-explosif, basé sur la grande croissance des machines maximales, et auto-limité basé sur la « foi » en l'existence d'une réelle maximalité.
Lorsque le test est validé, si l'enveloppe maximale provisoire numéro *n* n'est pas la valeur *M(n)*, cela signifie que l'émergence du très grand nombre E'_m n'a pas stabilisé M(n) et donc qu'il existe des zones de stabilités immenses et étalée dans lesquelles ne se produisent aucune apparition d'enveloppe, c'est-à-dire des croissances maximisée sur des échelles inimaginables qui sont en dessous d'une simple croissance locale. Cela laisse pantois sur l'idée de croissance maximale.
En effet, le problème étant de savoir si toutes les enveloppes M(n) pour n<m sont stabilisé, il faut aller très loin pour tenter de garantir cette stabilisation. Or quoi de plus grand qu'une enveloppe maximal
Rappelons que l'immensité des enveloppes maximale ne provient pas de la complexité ridicule du petit programme récursif de définition des Ei, mais de la complexité présente dans la liste et l'exécution de toutes les machines de Turing, on obtient donc une suite d'une croissance gigantesque définie à l'aide de peu de concept. (On aurait donc un petit programme capable de généré une complexité immense, ce mystère est discuté dans l'article « paradoxe de Kolmogorov »).
La question est de savoir si ce programme peut ou non apporté un arrêt effectif. Il est assez évident qu'on ne pourra pas en produire de preuve et qu'on pourra même produire la preuve qu'on ne peut en produire de preuve (à partir d'arguments -en rupture- de maximalité). Cependant ce petit raisonnement donne le vertige dans tous les cas : soit il donne une fonction d'arrêt effective, soit il

montre que la croissance maximale a quelque chose de proprement vertigineux. Cela dit, pensons bien que le petit bout de programme ajouté au programme initial pour tenter d'aller suffisamment loin, consiste à ajouter relativement peu de « code » (relativement peu d'états supplémentaires pour une machine de Turing). Or cette nouvelle machine fera beaucoup plus d'étape que la précédente. Cela signifie qu'en ajoutant très peu de code, on démultiplie le nombre de pas de façon gigantesque. Ce qui tant à confirmer l'idée qu'il existe de tel saut gigantesque et donc d'une croissance locale capable de surpasser une croissance très rapide à très longue distance.
Observons alors la « contradiction » en imaginant que ce processus donne la réponse en un temps fini. La réponse sera toujours 'non-arrêt', car il n'est pas prévu que le programme s'arrête. L'arrêt ou plutôt l'interruption est due à une borne qui stoppe en affirmant l'impossibilité théorique de produire une autre réponse que 'non-arrêt'. Et l'on voit apparaitre la confusion entre la réponse donnée par l'arrêt qui est 'non-arrêt' et l'arrêt effectif dû à une interruption. Ensuite, il faut inverser la valeur d'arrêt produisant la réponse 'arrêt'. A toutes les strates, on retrouverait le même processus, sauf que la première strate interrompt tout bien avant que les strates inférieures qui n'ont aucune existence pour le programme ne se finalisent. A y réfléchir deux secondes, il n'y a aucune raison pour que la strate supérieure donne sens aux strates inférieures, c'est notre lecture interprétative qui lui donne sens. Il n'y a plus de contradiction.

Constat final
En conclusion, on pourra retenir que l'« existence » d'un algorithme d'arrêt ainsi que le concept d'arrêt reposent sur une lecture bien subjective. Mais retenons que la fonction que nous avons fournie est en droite file avec les pratiques des mathématiques classiques, elle est même encore plus constructive. Maintenant, elle s'accorde moins avec les mathématiques constructives qui veulent accéder à une réponse en connaissance de cause : La construction a beau être de nature complètement constructive, l'interprétation finale ne l'est pas (comme dans le théorème de Gödel).

Si on met de côté le prolongement passablement discutable, on dispose d'une situation assez rare en mathématique classique : un procédé itératif constructif qui produit une valeur univoque (oui ou non) qui arrive avec « certitude », mais on ne sait pas quand. On dira même dire qu'« on ne peut pas savoir quand ». Cela permet de produire une foule de situations non intuitives à l'intérieur des mathématiques classiques. L'assurance d'« existences » avec une impossibilité d'accès promet de construire bien des objets cocasses. On pourra construire de tels exemples à partir des raisonnements non constructifs des mathématiques classiques : par exemples en appliquant le théorème des valeurs intermédiaires à une fonction dont les décimales sont construites sur cette réponse, on pourra produire constructivement une limite qui existe mais qui est impossible à déterminer, etc... Bien sûr, tout cela ne repose que sur des regards conceptuels en rupture qui nous plongent dans la subjectivité parce qu'ils affirment l'existence de choses essentiellement inaccessibles (on se croirait presque dans un cours de théologie). Maintenant ils ont beau être subjectifs, ces concepts ne sont pas forcément impertinents, ils permettent de raisonner avec des concepts « compatibles » sur des réalités qui sont inaccessibles en elles-mêmes. Il suffit de « s'en donner le droit » pour interpréter les processus avec un certain « réalisme », même s'il est subjectif. C'est le principe de l'analyse en rupture conceptuel.
A l'inverse, tout cela nous impose aussi un recul, une prise de distance sur les affirmations péremptoires issues des choix conceptuels en rupture. En particulier, toute preuve d'impossibilité qui repose sur un raisonnement en rupture n'est pas forcément verrouillée de toute part. C'est typiquement ce que nous avons fait en produisant une fonction d'arrêt constructive malgré le théorème d'arrêt. En créant des concepts plus fins, d'autres interprétations sont possibles... Cette remarque transforme en passoire toute prétention philosophique issue de théorèmes mathématiques. Est-ce vraiment surprenant, quand on voit les mathématiques comme de l'univocité interprétée ? L'univocité est neutre, elle peut être vue comme un constat de règle mécanique. Que pourraient dire des manipulations binaires ou un simple graphe sur la vérité ou la vie humaine. L'interprétation possède toujours une dimension de liberté. Avec un peu d'imagination, il est possible de recoller ensemble beaucoup de choses a priori étrangères. Les mathématiques classiques ne s'en privent pas comme dans les manipulations de l'infini.

Au final, il est amusant, et même rassurant, de voir que des concepts subjectifs produisent des conclusions relativement pertinentes sur un mode d'interprétation très subjectif : « la fonction d'arrêt est impossible car… ». En effet, même si on peut subjectivement rendre possible une telle fonction, elle semble manifestement impossible à garantir de façon strictement constructive. En dehors de tout raisonnement en rupture, c'est d'abord à cause de la complexité illimitée qu'est censée mesurer une telle fonction alors qu'on a aucune idée de ce que peut être la complexité... Maitriser la complexité par un concept simple semble manifestement dérisoire. Au travers de ces discours, on a découvert la dimension typiquement humaine et interprétative de concepts d'apparence simple et univoque ; comme l'est la structure binaire de l'arrêt. Tout cela parle de la grande subjectivité du concept d'arrêt, et de bon nombre de ses congénères…

Navenne – 2014 (révision 2019)

Michaël Klopfenstein