**Fondements de l'informatique, à la croisée des mathématiques, de la logique et de la linguistique**

Gérard Huet

*Colloque sur l'enseignement philosophique et les sciences*

*Journée du 13 novembre 2013*

Je vais essayer dans mon propos de défendre le point de vue que l’informatique est une science. C’est un exercice difficile, il faut toujours commencer par des précautions oratoires, que l’informatique n’a rien à voir avec la notion d’ordinateur par exemple, contrairement aux idées reçues. Bien sûr, les ordinateurs ont permis le développement de l’informatique en tant qu’outils. Mais la technologie des ordinateurs électroniques s’appuie essentiellement sur la physique. Par exemple, la fabrication des circuits des micro-processeurs, des mémoires et des contrôleurs, c’est un problème de physique du solide, qui n’a pas directement à voir avec le traitement de l’information et l’informatique. Une autre idée communément répandue est que l’informatique, c’est ce tsunami planétaire nommé Internet qui bouleverse complètement le champ social. Il faut laisser de côté cela si l’on veut comprendre l’aspect scientifique de l’informatique. Aussi je ne parlerai aucunement des considérations sociales autour de l’informatique, puisque l’exposé de Stéphane Grumbach y est consacré.

Les informaticiens revendiquent l’informatique comme étant d’une certaine manière une discipline autonome. Ce n’est pas si évident que cela. J’estime qu’en quarante ans de recherches en informatique j’ai essentiellement fait des mathématiques. L’activité de recherche en informatique est en effet par essence de nature mathématique. Alors ce sont des mathématiques un petit peu particulières peut-être, ce sont des méthodes mathématiques considérées comme plutôt marginales par la communauté mathématique, par rapport aux grands programmes ouverts de leur discipline. Lorsque je veux être provocateur, je dis que l’informatique c’est comme les mathématiques, simplement, c’est plus précis et c’est plus rigoureux. Jean-Pierre Kahane sait que j’aime bien taquiner les mathématiciens, mais les philosophes parmi vous doivent avoir les cheveux qui se dressent sur la tête : plus formel que les mathématiques, quelle horreur, ou quel blasphème. Je vais donc devoir essayer d’expliquer un petit peu cela. Mon collègue Aigrain nous a parlé de programmes. En effet, les programmes informatiques sont d’une certaine manière l’objet essentiel d’étude. Qu’est-ce qu’un programme ? Un programme c’est un texte formalisé qui peut être interprété comme une séquence d’opérations menant à un calcul effectif. Que ces opérations soient les opérations d’une machine physique ou d’une machine virtuelle ou même d’un humain. Ces textes sont écrits dans des langages formalisés qu’on appelle des langages de programmation, et ils peuvent être analysés comme des objets mathématiques en soi, que l’on appelle les algorithmes. En fait, c’est une investigation qui pré-date les ordinateurs, puisque les bases de la calculabilité ont été établies par des logiciens comme Turing, Gödel ou Church dans les années 30.

La programmation fut au début la simple itération d’opérations élémentaires d’une machine physique effectuant des modifications d’état d’une mémoire interne. Puis se dégagea la notion de langage permettant de définir et d’exécuter des opérations plus complexes, d’abstraire des calculs dans des procédures, de représenter des données dans des structures complexes. Le texte des programmes devint une espèce de grande formule mathématique exécutable. Récemment, la notion de « programmation fonctionnelle » a poussé ce point de vue en mettant au premier plan la notion de fonction. L’idée est de mettre au cœur du dispositif une algèbre fonctionnelle très simple mais de grand pouvoir expressif, vue comme espèce d’ossature commune à tous les langages de programmation. Cette algèbre fonctionnelle est encore peu connue, alors qu’elle remonte aux travaux du logicien Alonzo Church en 1940. Dans notre jargon on parle de lambda calcul. En fait, parler de lambda calcul c’est un peu mettre la charrue avant les bœufs, c’est parler d’un calcul avant de parler de l’algèbre sous-jacente, la fameuse algèbre fonctionnelle. Celle-ci est extrêmement simple. Comparons avec une notion élémentaire d’algèbre universelle, la structure de groupe. On postule une opération de groupe, binaire, une opération inverse unaire et une constante, l’élément neutre. C’est ce qu’on appelle la signature de la structure de groupe. De manière analogue, l’algèbre fonctionnelle postule trois opérateurs : abstraction de fonction (unaire), application de fonction (binaire), et on se donne comme constantes tous les entiers naturels, vus comme paramètres permettant d’exprimer l’argument formel de la fonction.

Laissez moi donner une illustration concrète. Par exemple, examinons un objet mathématique comme le polynôme x2 + 2x + 1. En fait, le statut de cette formule est un petit peu ambigu. Vous pouvez y penser comme dénotant une valeur réelle par exemple, pour une valeur réelle du paramètre x. Mais vous pouvez également y penser comme dénotant la fonction qui associe à la valeur x de son argument le résultat x2 + 2x + 1. Ainsi, il y a une espèce d’ambiguïté de cette notation, et suivant la manière dont ce polynôme va être placé dans le discours mathématique, il va dénoter un entier ou une fonction. Dans l’algèbre fonctionnelle, on distingue l’expression polynomiale x2 + 2x + 1 et son abstraction fonctionnelle explicite, notée **λ**x.x2 + 2x + 1, la fonction polynomiale correspondante, appelons-la F. On peut maintenant appliquer cette fonction F à un argument X, et l’expression correspondante F(X) dénote son résultat, c’est-à-dire X2 + 2X + 1. Ceci illustre le constructeur de l’algèbre appelé abstraction fonctionnelle, et noté λ : il capture la variable formelle x du polynôme, et forme la fonction polynomiale.

Dans l’exemple de la structure de groupe, on considère les lois du groupe, exprimées par des axiomes équationnels postulés comme des identités sur les opérations. Par exemple, x\*I = x exprime que l’élément I est neutre à droite pour l’opération \*. Une autre loi est l’associativité de l’opération \*. Dans notre algèbre fonctionnelle, il y a une loi unique, qui est tout simplement la règle de substitution :

 (λx.E[x] X) = E[X].

Vue de gauche à droite, cette loi devient une règle de calcul, qui permet de « simplifier » l’application d’une expression formelle explicitement fonctionnelle, en substituant à son argument formel x l’argument actuel X. En termes de programmeur, on « appelle » la procédure E avec la valeur X pour son paramètre x. Ceci semble si simple qu’on ne peut a priori imaginer que les calculs les plus compliqués puissent se réduire à l’enchaînement de cette loi de substitution qui semble juste tautologique. Il y a des petits détails techniques tout de même, ce n’est pas si simple que ça paraît. Par exemple, comment ne pas s’emmêler les pinceaux avec les noms des différentes variables formelles des fonctions imbriquées. C’est un problème bien connu en logique, avec la notation des quantificateurs, mais aussi avec la notation ensembliste { x | P(x)}, et puis avec les notations pour les sommes, les intégrales ou pire les dérivées partielles. La notation fonctionnelle permet de résoudre ce problème de nommage d’indice de liaison uniformément une bonne fois pour toutes. Toutes les notations spécifiques avec des variables liées deviennent alors des cas particuliers d’expressions fonctionnelles. Le problème a été complètement résolu en 1967 par le mathématicien Nicolaas de Bruijn, avec une notation d’indice arithmétique permettant d’éviter la confusion des noms.

Toutefois cette notation algébrique des expressions fonctionnelle n’est pas utilisable commodément dans le discours mathématique destiné à des humains, elle est réservée à l’implémentation de bas niveau, où l’indice de de Bruijn correspond à l’index dans la pile bien connu des compilateurs. Il en résulte que cette structure fondamentale d’algèbre fonctionnelle est encore très peu connue même des mathématiciens, et que le terme « lambda calcul » n’évoque généralement aux non spécialistes qu’un galimatias illisible de symboles, aux propriétés arcanes connues seulement de quelques rares experts.

C’est pour pallier cette difficulté que je me suis bien gardé d’annoncer d’emblée que j’allais vous parler de λ-calcul, et que j’ai dégagé le concept d’algèbre fonctionnelle comme un diamant caché dans la gangue d’un formalisme abscons. Evidemment, la terminologie pose question. Comment puis-je m’arroger le droit de parler d’algèbre fonctionnelle, sans me référer le moins du monde à la notion mathématique classique de fonction ?

Bien sûr, les mathématiciens ne nous ont pas attendu pour manipuler le concept de fonction. Plus exactement, jusqu’au 18° siècle, les fonctions en mathématiques étaient plutôt comme des algorithmes, avec une notion de calcul, et elles n’étaient pas des objets mathématiques en soi, c’était plutôt des éléments du discours. Puis vint l’Analyse, puis la Théorie des Ensembles, puis la camisole Bourbaki, et les fonctions ont été priées de rentrer dans le rang, enfermées dans le rôle secondaire de relations univoques. Je me souviendrai toujours de mon premier cours de mathématiques en école d’ingénieur. À peine étions-nous assis, le professeur demanda à l’amphi de donner la définition de ce qu’est une fonction. Tous les étudiants évoquant une notion de calcul furent traités d’ânes, jusqu’à ce que le fayot de service ne donne la bonne définition, un ensemble de paires <x,y> tel que pour tout x, il existe au plus un y tel que <x,y> appartienne à l’ensemble. C’est beau comme l’antique. Remarquez que la valeur de cette fonction en x, lorsque y existe, c’est y, obtenu instantanément. Il n’y a pas de dynamique, tout est sur la table. Il m’a fallu des années pour comprendre que j’avais été endoctriné par un bourrage de crâne ensembliste niant au départ la notion même d’algorithme.

En effet, ces fameuses fonctions mathématiques, en tant qu’objets mathématiques encodés au forceps dans le carcan ensembliste, sont marquées d’une faute originelle, l’extensionalité. Si f et g ont les mêmes valeurs pour tout argument de leur domaine commun, alors elles sont égales. C’est-à-dire in-dis-tin-gua-bles. Dans la nuit des fonctions ensemblistes, tous les algorithmes sont gris. On ne peut pas distinguer deux algorithmes, deux programmes qui vont calculer de manière différente peut-être, mais au bout du compte avoir le même résultat. Donc il faut bien comprendre que notre calcul fonctionnel ne parle pas de cette abstraction de fonction au sens de la théorie des ensembles, qui n’est que la relation entre les entrées et les sorties indépendamment de leur calcul, mais parle plutôt d’une autre notion plus intentionnelle, les algorithmes. Autrement dit, on fait attention à la manière dont la fonction est présentée par un programme, par exemple, par une suite de réécritures sur des formules. C’est pour cela que l’on parle d’algorithmes plutôt que de fonctions, et les algorithmes sont le premier objet d’étude de l’informatique.

Pour illustrer la différence, songeons à quoi étaient occupés principalement les ordinateurs  dans les années 70. Essentiellement à faire du tri. Les ordinateurs étaient utilisés à gérer des bases de données, à faire des calculs bancaires… On devait trier de grandes bases de données, et le tri a été un programme de recherche important pendant des années. Si vous expliquez à un mathématicien ce qu’est un problème de tri, par exemple de trier en ordre croissant une séquence finie d’entiers, il est probable qu’il n’y verra qu’une trivialité, qu’il saura résoudre instantanément. Par exemple, en engendrant toutes les permutations de la liste de départ, et en l’inspectant jusqu’à trouver la permutation dans l’ordre. Le problème c’est que ce processus est exponentiel, en temps et en mémoire. Si on essaye le procédé sur la base de données des immatriculations de voitures en France, mettons 50 millions d’entrées, on parle de factorielle 50 millions, nombre supérieur au nombre de particules dans l’univers. Il nous faut donc inventer des algorithmes un peu plus malins, et en comprendre les mérites en évaluant leurs temps de calcul et leur empreinte mémoire. Par exemple, Sir Tony Hoare inventa un algorithme de tri rapide appelé Quicksort, et qui fit sa gloire. Les mathématiciens n’ont pas l’habitude de cela, pourquoi ? Parce que tous les algorithmes de tri calculent la même fonction ensembliste, et que le théorème mathématique sous-jacent, l’existence de cette fonction, est une trivialité dont la preuve n’est pas importante. Pour nous, chaque algorithme correspond précisément à une preuve, et c’est cette preuve qui distingue un bon algorithme d’un mauvais. D’ailleurs on nomme les algorithmes et on célèbre leurs auteurs, alors qu’en mathématiques, on nomme les théorèmes, mais on ne nomme pas en général les preuves, qui sont des objets jetables.

Cette distinction est fondamentale. Derrière la pratique mathématique usuelle, il y a un principe épistémologique qui n’est écrit nulle part ni consciemment internalisé. C’est le principe de la non-pertinence des preuves. En effet, ouvrons un livre de mathématiques. On y trouvera un discours formalisé avec des axiomes, des définitions, et des théorèmes. On justifie les théorèmes par un morceau de rhétorique plus ou moins bien explicité, on pourra juger ces « démonstrations » plus ou moins élégantes, mais les preuves peuvent être effacées, et seuls comptent les énoncés des théorèmes, qui peuvent servir de lemmes intermédiaires aux théorèmes suivants, tout-à-fait indépendamment de leurs preuves. Il y a donc là un principe d’abstraction important que j’énonce comme étant la non-pertinence des preuves.

Si l’on écrivait un ouvrage d’algorithmique sur ce modèle, le chapitre du tri se réduirait à un théorème trivial, et on jetterait le bébé algorithme avec l’eau du bain de la démonstration considérée comme non essentielle. En fait, un algorithme de tri est essentiellement l’ossature de la démonstration qu’il vérifie la spécification de trier son argument. À chaque démonstration correspond un algorithme, qui a ses mérites propres, qui transcendent le simple fait que la fonction entre ses entrées et ses sorties est la bonne.

En fait ce principe de non pertinence est en voie de régression. On en voit les prémices dans des bizarreries de rédaction du discours mathématique. Par exemple, en théorie des nombres, il y a le Postulat de Bertrand, un résultat de densité des nombres premiers : pour tout nombre premier, il existe un nombre premier plus grand, mais inférieur à son double. Pourquoi n’est-ce pas le Théorème de Bertrand, car enfin c’est bien un théorème, et pas un axiome, comme la terminologie de postulat le suggère ? La réponse, c’est qu’on souhaite honorer Bertrand, qui a correctement postulé la conjecture, en la vérifiant jusqu’à 3 millions, et qui en a déduit des conséquences importantes. Mais la démonstration générale ne vint bien plus tard qu’avec Tchebychef, qui prouva un théorème très légèrement plus général. C’est ce qu’expliquent Hardy et Wright dans leur bible du domaine « An Introduction to the Theory of numbers ». En fait, l’énoncé de leur théorème 418 correspond bien à la conjecture de Bertrand, et ils en donnent une preuve plus élégante que celle de Tchebychef, et due à Erdös ! On voit déjà bien là que si l’on veut rendre à César ce qui est à César, il faut prendre au sérieux les preuves comme objets de première classe.

Les philosophes intéressés par l’épistémologie connaissent bien ce problème, qui est exacerbé par la mouvance des définitions, qui deviennent plus précises au fil du temps. Cette problématique a été magistralement développée dans « Preuves et réfutations », une perle de rhétorique autour de la conjecture d’Euler due à Imre Lakatos, et qui est malheureusement peu connue sauf de spécialistes.

Une autre bizarrerie intéressante est l’ « Hypothèse du continu », qui postule qu’il n’existe pas d’ensemble de taille (au sens du cardinal de la théorie des ensembles) intermédiaire entre le dénombrable (les entiers) et le continu (les réels). C’est tout de même relativement important d’un point de vue philosophique, car après tout les entiers et les réels sont les deux mamelles de Mathématique. Si l’on croit que Zermelo-Fraenkel est l’alpha et l’oméga de ses fondements, on doit pouvoir se prononcer sur cette hypothèse. Alors, est-ce que c’est vrai, ou est-ce que ce n’est pas vrai ? Hé bien, le statut de la chose est peu compliqué. On n’a qu’une réponse de Normand : p’têt ben qu’oui (Gödel), p’têt ben qu’non (Cohen). Mieux, c’est « comme tu veux tu choises ». Dans un sens, l’hypothèse est bonne, et son contraire aussi. On me dira que je fais encore de la provocation, mais c’est exactement le statut mathématique de la chose. On peut s’appuyer sur ce principe, dans le sens où il peut être ajouté aux axiomes standards de théorie des ensembles sans mettre davantage en péril sa cohérence présumée. Mais on peut aussi effectivement prouver son contraire, en postulant un cardinal inaccessible qui sera le témoin du fait que l’on peut « forcer » un ensemble à s’insinuer dans l’intervalle entre le dénombrable et le continu. Le problème, maintenant, c’est qu’on ne peut pas admettre des mathématiques s’appuyant sur l’hypothèse en coexistence pacifique avec des mathématiques s’appuyant sur son contraire, au risque de voir tout s’effondrer.

Comme on le voit sur cet exemple, les problèmes de fondement exigent de comprendre très précisément les preuves admissibles dans tel ou tel système axiomatique, sans certitude d’ailleurs de la cohérence absolue d’un tel système, en vertu des résultats de Gödel. Mais tout raisonnement mathématique n’a pas immédiatement un contenu calculatoire, et même des axiomes considérés comme anodins en logique classique posent un problème d’interprétation constructive. Donnons une fable pour l’illustrer.

Je vais vous raconter une histoire montrant la différence entre un informaticien et un mathématicien. D’abord, on imagine un mathématicien qui teste un nouvel étudiant sur ses capacités de raisonnement, et qui lui donne le problème : « Montrer qu’il existe X et Y irrationnels tels que X à la puissance Y soit rationnel ». L’étudiant rentre chez lui, réfléchit sur irrationnel, tout de suite il pense à √2 irrationnel. Il se dit que peut-être par chance √2 à la puissance √2 est rationnel, auquel cas il aurait la réponse avec X=Y=√2. Par contre, il est plausible qu’en fait ce nombre soit irrationnel. Mais en ce cas, en l’élevant à la puissance √2, il obtient √2 au carré, c’est-à-dire 2, et dans ce cas il a la réponse aussi. Il est tout content, va prendre une bière, et demain le Professeur sera content. En parallèle, Google vient d’embaucher un nouvel informaticien, et l’ingénieur en chef lui soumet un problème qui se pose dans une application au traitement d’images – il faut trouver deux paramètres à fournir au logiciel de rendu d’images, disons X et Y, tels que X et Y soient irrationnels mais que X à la puissance Y soit rationnel. Le jeune informaticien, qui a un bagage similaire au jeune mathématicien, va faire la même séquence de raisonnements. Pourtant, elles ne l’aideront en rien, il ne va pas savoir quoi faire de l’alternative. Il ne peut pas s’en tirer à aussi bon compte, et revenir vers son patron en disant voilà, le programme c’est : « Si √2 à la puissance √2 est rationnel, on retourne X = √2 et Y = √2, sinon, on retourne X = √2 à la puissance √2 et Y = √2 ». Il ne peut pas espérer que l’ordinateur trouvera la réponse tout seul, et son patron le traitera d’incompétent. Evidemment, l’une des deux alternatives est correcte, et il en existe sans doute une preuve quelque part, mais là ça n’est plus une trivialité.

Alors, où est la différence ? C’est qu’il est très facile de prouver l’existence de X et Y répondant à la question, en composant les deux branches de l’alternative avec l’axiome du tiers exclu, mais cela ne nous donne pas pour autant les valeurs concrètes de X et Y attendues par le logiciel de traitement d’images.

Ces questions de constructivité des raisonnements mathématiques ne sont pas vraiment nouvelles, elles étaient déjà soulevées par Brouwer il y a un siècle, mais à l’époque on les a traitées par le mépris, et on a relégué les méthodes constructives à un ghetto appelé par dérision « intuitionnisme », comme si ces procédés participaient d’un psychologisme sentant le soufre. Depuis l’arrivée de l’informatique, et de logiciens comme Per Martin-Löf intéressés à la modéliser par des méthodes constructives, un nouvel essor de la théorie de la démonstration se dégage sous l’intitulé « Théorie des types », apte à expliquer les mathématiques constructives régissant la programmation.

En fait le tiers exclu est une espèce de poison qui s’est introduit subrepticement dans les Mathématiques, dont le discours standard ressort à la logique classique. Logique classique dont le tiers exclu est la pierre angulaire. Par exemple, ouvrons Bourbaki au hasard. On y trouvera des « Propositions », exprimées sous la forme d’une proposition logique contenant des variables libres (Soient X un espace compact, R une relation d’équivalence dans X, C son graphe, etc.) et énonçant « Les conditions suivantes sont équivalentes : », le tout suivi d’un certain nombre d’assertions , considérées comme des paraphrases de la même notion et nommées a), b), c), etc. Suit une preuve que a) implique b), b) implique c), c) implique a) typiquement. C’est élégant, on encapsule des formulations supposées équivalentes d’une notion, et on obtient une « définition » de la notion reflétant ces divers points de vue, et utilisable sous divers angles d’attaque. Le problème, c’est qu’on n’attache pas d’importance aux procédés logiques intervenant dans les preuves des implications mutuelles. Notamment, l’axiome du tiers exclu peut s’y glisser, et ainsi diluer des notions non équivalentes d’un point de vue plus fin, apportant un élément de flou dans la définition du concept établi. On voit bien ici le côté pernicieux du principe de non-pertinence : ces définitions polluées vont à leur tour contaminer les développements les utilisant, sans traçabilité possible, avec l’indécidable mélangé au calculable.

Revenons justement au calculable. J’ai commencé mon discours en présentant le calcul fonctionnel, distinguant les algorithmes des fonctions extensionnelles, et proposant une notion de calcul universel. Cette notion de calcul introduit une dynamique dans l’élaboration du résultat d’une fonction. En particulier, les calculs peuvent ne pas terminer, et c’est là un aspect essentiel. Il est notamment essentiel pour justifier la complétude du langage de programmation au sens de Turing, c'est-à-dire du fait que l’on dispose d’un formalisme qui a le pouvoir de définir toutes les fonctions calculables. Ceci est obligatoirement au prix de l’introduction de notations pour des programmes qui peuvent ne pas terminer pour certaines valeurs de leurs données. Il n’y a pas une théorie des fonctions calculables, il y a une théorie des fonctions partielles calculables, et c’est toute la dynamique du calcul qu’il faut examiner avec soin pour être sûr que les définitions n’introduisent pas des cercles vicieux. Il convient donc de garantir qu’un programme va terminer sur des données qui sont bien configurées. De cette problématique est issue une famille de formalismes appelés systèmes de types ; ils viennent se greffer comme des décorations sur les représentations fonctionnelles aux fins de discipliner ce calcul pour qu’il soit un petit peu moins sauvage et par exemple d’assurer la terminaison des calculs. On évite ainsi des paradoxes comme le combinateur Ω=λx.x(x). En effet, l’expression Ω(Ω) mène à calcul par la règle de substitution dont j’ai parlé. Le résultat de ce calcul, c’est le terme x(x), dans lequel x est remplacé par Ω, c’est à dire l’expression de départ Ω(Ω), menant à un calcul infini. Ce problème est analogue au paradoxe de Cantor dans une théorie des ensembles naïve permettant de noter l’ensemble des ensembles qui ne se contiennent pas eux-mêmes. Dans le calcul fonctionnel, le paradoxe se traduit par un programme qui ne termine pas. Qui ne termine pas et qui ne produit d’ailleurs aucune approximation de résultat utile. L’utilisation de types permet de pallier cet inconvénient, et de limiter le pouvoir expressif des programmes pour qu’ils terminent, qu’ils ne provoquent pas de corruption des données, ni d’instruction illégale d’une machine d’exécution. On a ainsi développé toutes sortes de formalismes de programmes typés garantissant une certaine dynamique d’exécution préservant les structures de données exprimées par les types.

Et puis un jour on s’est rendu compte que cela avait été déjà fait dans un domaine connexe, la théorie de la démonstration en logique mathématique. Notamment, Gentzen avait défini dans les années 30 un formalisme de preuves appelé déduction naturelle, et qui n’est autre que notre calcul fonctionnel, muni de types qui sont les propositions de la logique considérée. La règle de calcul s’y appelle coupure, et le résultat de terminaison de ce calcul s’appelle l’élimination des coupures, théorème fondamental puisqu’il induit la cohérence de la logique. L’isomorphisme entre la théorie de la fonctionnalité et la théorie de la preuve n’a été compris que dans les années 80, et c’est un exemple rare de développement du même matériau mathématique en deux théories étanches sur une longue période. Cet isomorphisme est connu comme la correspondance de Curry-Howard-de Bruijn, en hommage à ces trois pionniers qui ont posé les bases du fondement logique de la programmation. C’est un lien fondamental entre la notion d’algorithme fonctionnel et la notion de preuve mathématique. Ainsi, nos algorithmes sont des manières d’exprimer des preuves mathématiques, et leur exécution préserve leur spécification logique. Nous sommes bien au cœur de la problématique : en rejetant le principe de non-pertinence des preuves mathématiques, nous leur apportons au contraire un sens calculatoire.

Les formalismes de théorie de la preuve un peu oubliés ont trouvé une nouvelle jeunesse avec la théorie des types et les langages fonctionnels, et ont pris une place considérable dans l’industrie du logiciel en apportant une sûreté de fonctionnement garantissable mathématiquement. Des assistants de preuve sont implémentés, permettant le développement de logiciels certifiés. Toute une industrie de recherche et développement est née de cette problématique, où la France est particulièrement bien placée, dans une grande mesure je crois à cause de l’éducation mathématique solide que nous avons reçue, au moins dans ma génération. Par contre coup, nos logiciels de preuves ont pu servir à la formalisation complète de développements mathématiques conséquents, classiques ou constructifs, ce qui a fini par attirer l’intérêt des mathématiciens « purs ».

Alors, à ce point de mon exposé, pouvons nous en déduire que l’informatique fondamentale, c’est exactement le champ des mathématiques constructives ? En fait, non, toutes les mathématiques nous sont utiles, il n’y a pas un sous-domaine précis délimitant les mathématiques utiles à l’informatique. Par exemple, si l’on consulte le Magnus Opus de Donald Knuth « The Art of Computer Programming », volume 1, on trouve à propos du premier programme présenté, énumérant les nombres premiers par le crible d’Erathosthène, une note de bas de page expliquant une optimisation apparemment anodine du programme, mais qui nécessite pour sa justification précisément le Postulat de Bertrand discuté plus haut, résultat dont la preuve nécessite tout un arsenal d’analyse non constructive. L’œuvre de Philippe Flajolet, sur l’analyse en moyenne des algorithmes, a donné lieu au développement de tout un domaine de combinatoire analytique, fertilisé de mathématiques classiques diverses et variées.

En fait, une nouvelle classification des Sciences est en train de se dégager. D’une part, la Physique au sens large, c’est à dire la Science de l’Univers matériel, aussi bien sciences de la nature, du vivant, des objets fabriqués par l’ingénierie, de la gestion de l’énergie, etc. Et d’autre part les Sciences de l’Immatériel, qui étudient les structures de l’information, de la communication, de la connaissance. Dans lesquelles on trouve l’informatique bien sûr, mais aussi l’automatique, la statistique, et les mathématiques appliquées à la modélisation, à la simulation, à l’optimisation, au calcul. Mais aussi la logique, la linguistique, la sémiotique, la modélisation de l’Economie, etc. Ces deux grandes divisions des disciplines scientifiques bénéficient des Mathématiques dans leur ensemble, et en retour contribuent à leur développement. Cette partition n’est pas rigide, et de nombreuses interactions existent entre le matériel et l’immatériel, comme l’illustre le développement récent d’une nouvelle algorithmique utilisant la mécanique quantique.

Alors, me direz-vous, si vous prétendez que l’informatique est une science à part entière, et même LA science de ces objets immatériels qui peuplent ce monde numérique en pleine croissance, comment faut-il l’introduire à l’école pour que nos jeunes puissent en bénéficier dans leur vie professionnelle ?

Je me garderai de faire des recommandations concernant la vache sacrée des programmes de l’Education Nationale. Il est un peu vain de prétendre bouleverser un équilibre fragile entre l’instruction des savoirs fondamentaux et une éducation scientifique à la page. Dire qu’il faut faire 2h d’informatique en seconde, et qu’en conséquence il faut d’urgence supprimer 2h des autres matières n’aboutira qu’à se faire battre des montagnes. Il y a à mon avis une manière plus consensuelle d’aborder l’informatique à l’école. D’abord, en intégrant son aspect ludique. Apprenons à nos jeunes à programmer à travers des clubs où ils apprendrons entre eux à concevoir et réaliser des jeux électroniques et des applications téléphoniques rigolotes, à rentrer dans des Olympiades sur le réseau où ils valoriseront leurs acquis tout en s’amusant. Sachons aussi utiliser l’aspect transversal de l’informatique pour en diffuser la méthodologie et les techniques dans toutes les disciplines.

On nous dit « il faut inculquer aux jeunes l’esprit scientifique ». Très bien, mais qu’est ce que ça veut dire au juste, au-delà d’une incantation un peu creuse ? Inculquer l’esprit scientifique ne se fait pas à coup de bourrage de crâne de connaissances scientifiques rebutantes, ce qui est au contraire la meilleure manière de faire fuir les élèves. De toutes façons, la science moderne est trop vaste et trop complexe pour que quiconque puisse tout connaître, on n’aura plus de Pic de la Mirandole, et c’est aussi bien. Par contre, on peut susciter la curiosité des élèves en mettant en valeur les figures de rhétorique développées par la science pour acquérir ces connaissances. Pour avoir prononcé ce terme de rhétorique devant vous, je devrais m’excuser, c’est un terme vieillot. Autrefois, il y avait des classes de rhétorique, et la notion de débat intellectuel était valorisée. Maintenant c’est terminé, on inculque des connaissances prédigérées, et la Science est imposée comme un prêche. On apprend par cœur des formules que l’on fait réciter, les exercices sont calibrés pour être résolus par application mécanique d’un cours bien saucissonné, l’esprit critique n’est pas encouragé. Ouvrons la fenêtre, discutons des méthodes qui permettent de raisonner droit, de comprendre comment poser des hypothèses, d’élaborer des conjectures, de chercher des contre-exemples. Ces méthodes sont transversales à toutes les matières enseignées, littéraires comme scientifiques. Il y a là un lien important entre philosophie et informatique, car la méthodologie informatique prolonge la rhétorique traditionnelle en tant que moyen légitime d’acquérir des connaissances. Ce sont ces préoccupations qui ont développé la logique, qui a finalement quitté la philosophie pour s’intégrer aux mathématiques, mais a perdu en passant sa finalité argumentative, qui est l’essence de la démarche scientifique.

Je voudrais illustrer mon propos avec deux exemples.

Le premier exemple, tiré de la physique élémentaire, c’est le problème des robinets, version moderne avec une machine à laver. Vous avez une machine à laver, de puissance 2 kw. Elle a un cycle de lavage d’une demi-heure, donc vous dépensez comme énergie 1 kwh. Combien cela coûte-t-il ? Vous regardez votre contrat EDF, ça coûte 8 ct d’euros le kwh, donc au bout du compte un cycle de machine, cela vous coûte 8 ct d’euros. Deux trivialités, mais derrière ces trivialités il se cache un procédé rhétorique commun. On a exprimé que la puissance c’était quelque chose qui s’intégrait par l’énergie. C’est une fonction du temps vers l’énergie dépensée. Vous pouvez ainsi re-conceptualiser votre lave-linge. Sa fonction n’est pas tant de laver le linge, sa vraie fonction au regard de la question posée c’est de dépenser de l’argent au fil du temps. On dégage une première fonction qui transforme le temps en énergie, et puis une deuxième fonction qui est, en gros celle de votre contrat EDF qui dit voilà, si vous dépensez tant d’énergie vous me devez tant d’euros. On peut d’ailleurs remarquer que la fonction de laver le linge est non pertinente de ce point de vue, puisque le cycle vous coûtera autant que vous mettiez du linge dans le tambour ou pas. On a juste composé ces deux petits processus fonctionnels pour calculer le prix d’un cycle de lavage de votre machine. Quand on décortique, qu’est ce qui s’est passé ? Il s’est passé qu’il y a eu un petit raisonnement, le premier étage a dit qu’une fonction du temps vers l’énergie (la puissance de la machine) peut s’appliquer au temps et fournir de l’énergie. Le deuxième étage dit qu’une fonction de l’énergie vers l’argent (le prix de cette énergie) produit de l’argent. Evidemment, au bout du compte, c’est de l’argent dépensé, de même que la machine ne produit pas de l’énergie mais en dépense, mais justement, cela permet d’expliquer aux élèves la différence entre système ouvert et système fermé, où l’utilisateur de la machine rentrera dans la boucle pour payer la facture EDF. Hé bien, maintenant reconnaissons ce procédé rhétorique utilisé deux fois pour les deux applications fonctionnelles. C’est tout bêtement le bon vieux Modus Ponens d’Aristote, ré-habillé par la correspondance de Curry-Howard-de Bruijn comme règle de typage de l’application de fonctions. Et la composition des deux fonctions correspond à la transitivité de l’implication. Plutôt que de se focaliser sur les règles de trois à utiliser pour faire les calculs, et qui dépendent d’une hypothèse de linéarité des fonctions particulières, on a enseigné en passant le calcul de dimensions, une notion fondamentale à inculquer en physique au lycée pour donner du sens aux formules.

Je vais prendre un autre exemple dans un domaine complètement différent, c’est l’analyse grammaticale dans la classe de français. Je ne sais pas si on fait encore beaucoup ça, mais de mon temps on décortiquait les phrases : toute phrase doit avoir un verbe, tout verbe doit avoir un sujet. Là, il y a un petit bout de raisonnement aussi. Comment est-ce que l’on obtient une phrase à partir d’un verbe ? Prenons d’abord un verbe intransitif. Un verbe intransitif a besoin d’un sujet pour exprimer son action. Donc, vous pouvez voir le rôle fonctionnel de ce verbe comme utilisant le syntagme nominal représentant le sujet pour construire la phrase représentant l’action. De même, un verbe transitif peut être vu comme une fonction qui prend son complément d’objet pour construire un syntagme verbal, se comportant comme un verbe intransitif. Vérifier que « le chat mange la souris » est une phase correcte devient un petit raisonnement où le verbe « mange » prend la souris pour construire « mange la souris », objet fonctionnel qui peut maintenant manger le chat, si je puis dire, pour donner la phrase. Le petit arbre de raisonnement, qui exprime la composition des deux fonctions en vérifiant leurs types, hé bien, c’est ce qu’on appelle l’analyse grammaticale de la phrase. Regardez de près, vous vous rendez compte que c’est exactement le même raisonnement que celui pour la machine à laver du cours de physique, avec deux étapes de modus ponens. L’analyse dimensionnelle devient l’analyse grammaticale. C’est important, en exhibant les procédés rhétoriques similaires on abstrait le raisonnement commun, pour lequel les deux disciplines fournissent des exemples concrets. Les deux exemples s’éclairent l’un l’autre, et on retient un procédé cognitif général qui peut servir pour toutes sortes d’autres investigations. En exhibant le procédé rhétorique commun, et en le réifiant dans deux disciplines supposées étanches l’une à l’autre, on apprend aux élèves que l’esprit scientifique transcende les matières enseignées et les présente comme des aventures intellectuelles cohérentes. Et puis, cela peut donner des idées. En classe de français on faisait de l’analyse grammaticale, mais on n’en faisait pas en classe d’anglais. Pourquoi ? Le même type de raisonnement s’applique, et on montre deux exemples du même phénomène, qui est ainsi mieux mémorisé. Par contre il y a des détails de grammaire qui ne sont pas les mêmes. Par exemple, en introduisant les paramètres morphologiques, on va pouvoir exprimer l’accord du verbe avec son sujet comme une contrainte sur les arbres d’analyse. En français comme en anglais. Par contre, l’adjectif est invariable en anglais, et donc ne s’accorde pas avec le nom qu’il qualifie. En mettant en lumière ces différences structurelles fondamentales, on éclaire les difficultés rencontrées par les élèves, les faux amis, les analogies erronées qui sont difficiles à déraciner. C’est important de le montrer en contraste avec le français. Parce que si vous leur apprenez l’analyse grammaticale du français, il y a une grande partie qu’ils vont pouvoir appliquer à l’anglais aussi, et les parties où cela ne s’applique pas, c’est justement les endroits où il faut faire attention à ne pas calquer d’une langue sur l’autre.

Alors introduire l’informatique au collège et au lycée, oui bien sûr, il va falloir le faire. Par l’encouragement de la pratique de la programmation, d’abord, mais dans un cadre périscolaire encadré plutôt qu’en enseignement magistral (et surtout en évitant la terminologie absurde de « codage » utilisée récemment par les médias). Ensuite, en faisant percoler la méthodologie informationnelle à travers les différentes disciplines – ce qui suppose un niveau d’échanges entre professeurs de disciplines différentes supérieur à zéro. Enfin en intégrant les méthodes mathématiques appropriées (combinatoire, récurrence, structures de listes, arbres et graphes, automates, algorithmes) dans un programme de mathématiques ré-ouvert à ses racines calculatoires.