Le logiciel, entre l'esprit et la matière

Xavier Leroy

Xavier Leroy

Le logiciel, entre l'esprit et la matière
Leçon inaugurale prononcée au Collège de France le jeudi 15 novembre 2018

Collège de France

Le logiciel, entre l'esprit et la matière

Leçon inaugurale prononcée au Collège de France le jeudi 15 novembre 2018

Xavier Leroy

Éditeur : Collège de France
Lieu d'édition : Paris
Année d'édition : 2019
Date de mise en ligne : 10 décembre 2019
Collection : Leçons inaugurales
ISBN électronique : 9782722605299

http://books.openedition.org

Édition imprimée
Date de publication : 24 avril 2019

Ce document vous est offert par Collège de France

Référence électronique
LEROY, Xavier. *Le logiciel, entre l'esprit et la matière : Leçon inaugurale prononcée au Collège de France le jeudi 15 novembre 2018* In : *Le logiciel, entre l'esprit et la matière : Leçon inaugurale prononcée au Collège de France le jeudi 15 novembre 2018* [en ligne]. Paris : Collège de France, 2019 (généré le 10 décembre 2019). Disponible sur Internet : <http://books.openedition.org/cdf/7681>. ISBN : 9782722605299.

Le logiciel, entre l'esprit et la matière

Leçon inaugurale prononcée au Collège de France le jeudi 15 novembre 2018

Xavier Leroy

1 Monsieur l'Administrateur,
Chers collègues,
Chers amis,
Mesdames et Messieurs,

2 Sur le site internet de France Culture, on peut lire ceci :

> Autrefois nommée « harangue », la leçon inaugurale prononcée par chaque professeur élu au Collège de France est à la fois la description de l'état d'une discipline et la présentation d'un programme de recherche. Bien que le nouveau titulaire soit aguerri, cet exercice demeure une épreuve formidable[1].

3 Au moment de commencer cette leçon inaugurale – ou devrais-je dire cette harangue ? –, « épreuve formidable » me semble très juste, mais « aguerri » très exagéré : « extrêmement ému » conviendrait mieux à mon état d'esprit actuel. Émotion face à l'honneur que vous me faites, chers collègues, et à la confiance que vous me témoignez en m'accueillant parmi vous. Émotion, aussi, de voir ma discipline, l'informatique, ainsi reconnue, et sa place au Collège de France ainsi confirmée.

4 L'informatique, comme pratique et comme science, a longtemps été ignorée par le système universitaire français[2]. Il faut attendre le milieu des années 1960 pour que l'informatique s'introduise dans l'enseignement supérieur et la recherche français ; 1980, environ, pour que les principales universités scientifiques françaises aient une unité d'enseignement et de recherche en informatique ; 1997 pour qu'un informaticien, Gilles Kahn, soit élu membre de l'Académie des sciences ; 2000 pour que le CNRS crée un département entièrement consacré aux sciences et technologies de l'information et de la communication ; et 2007 pour que l'informatique entre au Collège de France, dernière étape vers la respectabilité universitaire.

5 C'est Gérard Berry, pédagogue hors pair et infatigable ambassadeur de la discipline, qui a introduit l'informatique au Collège de France, d'abord sur la chaire annuelle Innovation technologique en 2007, puis en 2009 comme premier invité de la chaire annuelle Informatique et sciences numériques créée en partenariat avec l'Inria, et, enfin, en 2012 comme titulaire de la première chaire permanente en informatique, intitulée « Algorithmes, machines et langages ». Pierre Bourdieu qualifiait le Collège de France de « lieu de sacralisation des hérétiques ». Il n'avait pas précisé que plusieurs tentatives pouvaient être nécessaires pour atteindre au sacré.

6 Cette chaire Sciences du logiciel que vous me faites l'honneur de me confier est donc la deuxième chaire permanente en informatique dans l'histoire du Collège de France. C'est la confirmation que cette discipline a toute sa place au sein de cette institution. C'est aussi un signe fort de reconnaissance adressé à tous les informaticiens – universitaires, industriels, libristes, amateurs – qui créent le logiciel et travaillent sans relâche à le rendre meilleur. Au nom de cette communauté, je remercie l'Assemblée des professeurs du Collège de France d'avoir créé cette chaire, et Pierre-Louis Lions d'en avoir porté le projet.

Une brève histoire du logiciel

7 Les premiers dispositifs mécaniques *programmables* apparaissent en Europe au cours du XVIII^e^ siècle : orgues de Barbarie et métiers à tisser de type Jacquard. C'est l'arrangement des pleins et des trous sur une carte perforée qui détermine l'air de musique ou le motif textile produits. Un même mécanisme – un même *matériel* – peut produire une infinité de résultats visuels ou sonores rien qu'en changeant les informations présentes sur la carte perforée – le *logiciel*. Les seules limites ici sont la longueur de la carte perforée et l'imagination de celles et ceux qui la préparent : les *programmeurs* et les *programmeuses*.

8 Il y avait là une idée forte qui aurait pu donner à réfléchir. Pourtant, elle est passée largement inaperçue et restée cantonnée aux ateliers de tissage et aux fêtes foraines pendant près de deux siècles. La vague d'automatisation qui a balayé l'industrie dès le XIX^e^ siècle puis la vie quotidienne après 1950 s'est appuyée sur des machines et des instruments d'abord mécaniques, ensuite électriques, puis électroniques, mais non programmables et construits pour exécuter une et une seule fonction.

9 C'est aussi le cas du calcul numérique, pourtant indissociable de l'ordinateur moderne dans l'imagerie collective. La calculatrice programmable a beau accompagner les lycéens depuis 1980, il n'y a rien de programmable dans les instruments de calcul largement utilisés jusque bien après la Seconde Guerre mondiale : de la machine d'arithmétique de Blaise Pascal (1645) au calculateur mécanique de poche Curta cher à Gérard Berry ; des machines tabulatrices qui permirent les grands recensements du XX^e^ siècle aux caisses enregistreuses de mon enfance ; des calculateurs de tir d'artillerie aux commandes de vol électriques des premiers avions supersoniques[3]. Même le calculateur « bombe », qui permit aux services secrets britanniques de casser le chiffre allemand Enigma pendant la Seconde Guerre mondiale, n'était pas programmable et devait être fréquemment recâblé. C'est d'autant plus étonnant que l'architecte en chef de ces « bombes », Alan Turing, avait quelques années auparavant construit les bases théoriques du calculateur programmable – la célèbre machine universelle de Turing,

dont nous reparlerons bientôt. Mais il y avait une guerre contre l'Allemagne nazie à gagner, et pas de temps pour réfléchir à la programmation des calculateurs.

10 Une exception : en 1834, le mathématicien britannique Charles Babbage imagine son « moteur analytique » (*analytical engine*), où une carte perforée à la manière des métiers Jacquard contrôle les opérations d'un calculateur mécanique complexe, le « moteur à différences » (*difference engine*[4]). Malgré un généreux financement du gouvernement britannique, la construction du « moteur » de Babbage échoue, car il est trop complexe pour la technologie mécanique de l'époque. En revanche, son amie Ada Byron, comtesse de Lovelace, écrit sur papier quelques programmes pour le moteur analytique, notamment un programme qui calcule la suite de nombres de Bernoulli, faisant d'elle la première programmeuse informatique de l'histoire[5]. Babbage a-t-il inventé l'ordinateur moderne ? Les avis sont partagés. En revanche, c'est bien le pionnier du projet de recherche moderne. Tout y est : un financement par une agence d'État, des objectifs trop ambitieux pour être réalisables, et un résultat accessoire (le premier programme de calcul numérique) qui se révélera de la plus haute importance – ledit résultat étant obtenu par une femme qui n'était même pas financée par le projet.

11 C'est seulement à la fin de la Seconde Guerre mondiale que s'impose l'idée du calculateur universel parce que programmable. Les travaux fondateurs de John Presper Eckert et John Mauchly (1943) et de J. von Neumann (1945) établissent l'architecture des ordinateurs modernes : une unité de calcul communiquant avec une mémoire contenant à la fois le programme qui pilote les calculs et les données sur lesquelles ils opèrent. Plusieurs prototypes sont réalisés dans des universités à la fin des années 1940 (ENIAC, Manchester Mark 1, EDSAC, EDVAC…), et les premiers calculateurs électroniques programmables que nous appelons ordinateurs aujourd'hui sont commercialisés à partir de 1952 (Ferranti Mark 1, Univac 1, IBM 701…)[6].

12 La suite de l'histoire est mieux connue. Les ordinateurs sont sortis des centres de calcul et ont envahi les usines, les bureaux, les domiciles, les moyens de paiement, les téléphones, les véhicules, les équipements médicaux, les appareils photo, les téléviseurs, l'électroménager, et jusqu'aux ampoules électriques qui sont maintenant « connectées » et prétendument « intelligentes ». Ils prennent une part toujours croissante dans notre vie quotidienne et dans celle de la cité, du divertissement à la gestion d'infrastructures essentielles et à la préservation de vies humaines[7].

13 Cette explosion de l'informatique doit beaucoup aux immenses progrès de la micro-électronique, qui débouchent sur une production de masse d'ordinateurs et de systèmes sur puce toujours plus puissants à coût constant, mais aussi à l'incroyable flexibilité du logiciel qui s'exécute sur ces systèmes. Le matériel devient une *tabula rasa* reprogrammable à l'infini. Ainsi, les sondes spatiales Voyager ont été régulièrement reprogrammées à distance pendant les quarante ans de leur traversée du système solaire. Libéré de presque toute contrainte physique, le logiciel peut atteindre une complexité invraisemblable. Un navigateur représente environ 10 millions de lignes de code ; le logiciel embarqué dans une voiture moderne, 100 millions ; l'ensemble des logiciels de Google, 2 milliards. Qu'un assemblage de 2 milliards de choses toutes différentes fonctionne à peu près est sans précédent dans l'histoire des techniques. Sans précédent non plus : la grande vulnérabilité du logiciel aux erreurs de conception et de programmation – les célèbres *bugs* – et à l'utilisation malveillante.

14 Je pourrais vous parler longuement encore des prouesses et des misères de l'informatique moderne, et continuer à vous éblouir avec des chiffres et à vous effrayer

avec des risques. Je préfère maintenant revenir sur les concepts fondamentaux de la programmation et l'histoire de leur apparition. Ce n'est pas dans le calcul numérique qu'il faut chercher leurs racines, mais dans une tout autre branche des mathématiques, à la frontière avec la philosophie : la logique.

Les fondations logiques

15 À plusieurs reprises, les mathématiciens et les philosophes se sont tournés vers le calcul comme source de raisonnements incontestables et accessibles à tous[8]. Dès les années 1670, Gottfried Wilhelm Leibniz rêve de représenter les concepts philosophiques par des symboles mathématiques, et d'identifier les règles de calcul symbolique qui permettent de raisonner sur ces concepts. Avec ce *calculus ratiocinator*, comme il l'appelle, les désaccords philosophiques pourraient se résoudre par simple calcul :

> *Quando orientur controversiae, non magis disputatione opus erit inter duos philosophus, quam inter duos computistas. Sufficiet enim calamos in manus sumere sedere que ad abacos, et sibimutuo (accito si placet amico) dicere : calculemus*[9].
> Alors, il ne sera plus besoin entre deux philosophes de discussions plus longues qu'entre deux comptables, puisqu'il suffira qu'ils saisissent leur plume, qu'ils s'asseyent à leur table de calcul (en faisant appel, s'ils le souhaitent, à un ami) et qu'ils se disent l'un à l'autre : « Calculons ! »

16 Le chemin est long de ce rêve de Leibniz à la logique mathématique moderne[10]. Cependant, cet impératif, *calculemus*, est resté comme un cri de ralliement chez les informaticiens : c'est notamment le titre d'une série de congrès sur le calcul symbolique. Calculons, mes frères ! Calculons, mes sœurs ! Il en sortira des vérités !

17 Dans la seconde moitié du XIX^e^ siècle, la logique fait d'énormes progrès, avec la formalisation du calcul propositionnel (George Boole, 1854) et du calcul des prédicats (Gottlob Frege, 1879), ainsi que la naissance de la théorie des ensembles (Georg Cantor, 1875-1884)[11]. Vers 1900, il devient envisageable de fonder l'intégralité des mathématiques sur la base d'une logique formelle. Cependant, des paradoxes logiques apparaissent, qui minent ce beau projet. Le plus célèbre est le paradoxe de Bertrand Russell (1903) : si on peut définir $A = \{x \mid x \notin x\}$, « l'ensemble de tous les ensembles n'appartenant pas à eux-mêmes », alors A appartient à A en même temps que A n'appartient pas à A. Cette contradiction interne rend la théorie naïve des ensembles *incohérente* et donc inutilisable comme logique mathématique.

18 Le bel édifice des mathématiques, qu'on s'imagine s'élançant vers le ciel, bien appuyé sur de solides fondations, semblable à la tour Eiffel, ressemblerait-il davantage à la tour de Pise, penchant dangereusement à cause de fondations défectueuses ? C'est la crise des fondements des mathématiques, qui va préoccuper quelques-uns des plus grands mathématiciens et philosophes au début du XX^e^ siècle[12]. En 1900, David Hilbert, dans sa liste de 23 grands problèmes mathématiques ouverts, mentionne en deuxième position le problème de la cohérence de l'arithmétique – un fragment important des mathématiques. Vingt ans plus tard, il formule ce que nous appelons aujourd'hui le « programme de Hilbert ». Il s'agit de formaliser l'arithmétique par un système déductif[13] et de démontrer que ce système satisfait trois propriétés essentielles :

- *cohérence :* il n'existe pas de proposition P, telle qu'on puisse déduire P et sa négation non-P (la logique ne contient pas de paradoxes) ;

- *complétude :* pour toute proposition *P* on peut déduire *P* ou non-*P* (la logique n'a pas le droit de dire « je ne sais pas ») ;
- *décidabilité* (*Entscheidungsproblem*, « problème de la décision ») : il existe un procédé systématique – un algorithme, disons-nous aujourd'hui – qui, étant donnée une proposition *P*, décide si elle peut être déduite ou non.

19 On voit ici que, tout comme Leibniz avec son *calculus ratiocinator*, Hilbert accorde une grande importance à la possibilité de calculer la véracité d'une proposition logique.

20 Tel un professeur du Collège de France qui « passe » sur France Culture, Hilbert popularise ce programme *via* une célèbre allocution radiodiffusée en 1930, qui se conclut par « *Wir müssen wissen* ; *wir werden wissen* » : « nous devons savoir ; nous saurons[14] ». Peu de temps après, nous sûmes… que le programme de Hilbert est irréalisable. En 1931, Kurt Gödel publie son célèbre premier théorème d'incomplétude[15], qui montre que toute axiomatisation cohérente de l'arithmétique contient un énoncé *P* tel qu'on ne peut déduire ni *P* ni non-*P*. En 1936, Alonzo Church et Alan Turing démontrent, indépendamment et suivant des approches différentes, les premiers résultats d'indécidabilité (du problème de l'arrêt d'un calcul), d'où il s'ensuit que le *Entscheidungsproblem* n'a pas de solution algorithmique[16].

21 C'est la fin du programme de Hilbert mais le début d'un nouveau savoir. L'informatique fondamentale est née de cet échec du programme de Hilbert, telle une herbe sauvage qui pousse sur les ruines d'un temple effondré. Pour démontrer l'incomplétude, Gödel montre comment représenter toute formule logique par un nombre entier. Aujourd'hui, on utiliserait plutôt une suite de « bits », 0 ou 1, et un codage plus compact que celui de Gödel, mais l'idée est bien là : toute information – nombre, texte, son, image, vidéo, formule logique, programme informatique, etc. – peut être codée par une suite de bits afin d'être transmise, ou stockée, ou manipulée par un ordinateur. Et pour montrer l'indécidabilité, Church et Turing caractérisent précisément ce qu'est un algorithme, créant ainsi la théorie de la calculabilité, chacun à sa manière. Turing formalise sa « machine universelle », un petit robot imaginaire qui déplace une bande et y lit et écrit des symboles, capturant ainsi l'essence de l'ordinateur moderne : le calculateur programmable à programme enregistré. Church développe son « lambda calcul », une notation algébrique centrée sur la notion de fonction, qui est le grand ancêtre des langages de programmation modernes.

22 Cette naissance de la théorie de la calculabilité est un moment si important dans l'histoire de l'informatique qu'il mérite une métaphore culinaire. On dit souvent qu'un algorithme, c'est comme une recette de cuisine. Pour expliquer un algorithme résolvant un problème donné, comme pour communiquer une recette de cuisine produisant un plat donné, le langage naturel suffit : nul besoin de formaliser mathématiquement la recette. Ce n'est plus le cas s'il nous faut raisonner sur toutes les recettes possibles et tous les plats qu'elles produisent. Ainsi en est-il du problème de la recette (*Rezeptproblem*) :

> Toute nourriture peut-elle être produite par une recette de cuisine ?

23 Pour répondre par la négative, il nous faut identifier une nourriture non « cuisinable », comme l'ambroisie de la mythologie grecque, puis démontrer qu'aucune recette ne peut la produire. Pour cela, il est nécessaire d'avoir une définition mathématique de ce qu'est une recette. (Par exemple, « sonner à la porte de Zeus pour lui demander un reste d'ambroisie » n'est pas une recette.) Ainsi, nous allons développer une *théorie de la « cuisinabilité »* qui sera plus générale et plus intéressante que le *Rezeptproblem* initial.

24 *Mutatis mutandis* et toutes proportions gardées, c'est une démarche similaire qu'ont suivie Church et Turing pour répondre négativement au *Entscheidungsproblem*. Qui plus est, les deux modèles de calcul qu'ils ont proposés, quoique d'inspirations bien différentes, sont équivalents entre eux, au sens où l'un peut simuler l'autre, et équivalents à un troisième modèle d'inspiration plus mathématique, les fonctions μ-récursives étudiées par Stephen Cole Kleene à la même époque. Une machine (la machine universelle de Turing), un langage (le lambda-calcul de Church), et une classe de fonctions mathématiques (les fonctions μ-récursives) s'accordent sur les fonctions qui sont calculables et les problèmes qui sont décidables. C'est la naissance d'une notion universelle de calcul, la complétude au sens de Turing. Des centaines de modèles de calculs sont connus aujourd'hui, des jeux mathématiques aux modèles inspirés du vivant en passant par l'ordinateur quantique, qui calculent tous les mêmes fonctions qu'une machine de Turing.

Vers l'efficacité : algorithmique et programmation

25 Quelque chose manque encore pour que les rêves de Turing et les lambda-obsessions de Church débouchent sur l'informatique moderne : un aspect quantitatif, complètement absent de la théorie de la calculabilité. Ainsi, un problème peut être décidable simplement parce que l'espace des solutions est fini, en testant les 2^N solutions possibles, même si cela épuiserait toute l'énergie de notre Soleil dès que N atteint 200 environ[17]. De même, un modèle de calcul comme le jeu de la vie de Conway peut être Turing-complet et pourtant parfaitement inadapté à la programmation.

26 La dernière étape qui mène à l'informatique fondamentale moderne est précisément la prise en compte de ces impératifs d'efficacité des algorithmes (*efficiency*) et de capacité à programmer ces algorithmes (*effectiveness*). D'un côté se développe l'algorithmique : la science de concevoir des algorithmes et d'en caractériser mathématiquement la consommation en temps, en espace, ou en énergie. De l'autre, la programmation de ces algorithmes – leur « implémentation » comme nous, informaticiens, disons – fait naître de nouveaux besoins : des principes de structuration et de composition des programmes ; des langages de programmation expressifs ; des sémantiques pour ces langages ; des techniques d'interprétation et de compilation pour rendre ces langages exécutables par le matériel ; des méthodes de vérification pour s'assurer de l'absence d'erreurs de programmation – le *bug* tant redouté. Ces savoirs, mi-empiriques, mi-mathématiques, sont autant de balises sur le long chemin menant de la spécification abstraite d'un logiciel à son implémentation effective. Ensemble, ils constituent le cœur des *sciences du logiciel* dans lesquelles s'inscriront mes enseignements et mes recherches au Collège de France.

27 Je ne parlerai pas davantage d'algorithmique dans cette leçon, et en parlerai peu dans mes premiers cours, tant cette problématique a été brillamment exposée par Gérard Berry et par nombre des professeurs qui se sont succédé sur la chaire annuelle d'Informatique et sciences numériques, notamment Bernard Chazelle en 2012, Jean-Daniel Boissonnat en 2016, Claire Mathieu en 2017 et Rachid Guerraoui en 2018. Ces professeurs ont su communiquer toute la richesse et la diversité de ce domaine, des algorithmes probabilistes à la géométrie algorithmique, des algorithmes d'approximation au calcul distribué. Je vais plutôt parler du chemin qui mène d'un

algorithme abstrait à son exécution concrète par la machine, et plus particulièrement des langages de programmation et de la vérification des programmes.

Les langages de programmation

28 Dans les films, on voit les informaticiens scruter des écrans remplis de 0 et de 1 qui défilent à toute allure et y reconnaître au premier coup d'œil le virus que les méchants sont en train d'envoyer. La réalité est assez différente. Le code informatique composé de 0 et de 1 existe bel et bien : c'est le *langage machine*, celui qui est exécuté directement par les circuits électroniques de l'ordinateur. Mais ce langage est totalement inadapté à la programmation, tant il est illisible et dénué de structure. Dès l'apparition des premiers ordinateurs, les programmeurs n'ont cessé d'inventer des langages de programmation de plus haut niveau, plus clairs, plus expressifs, plus « parlants », afin de faciliter l'écriture mais aussi la relecture et l'évolution des programmes. Parallèlement, ils ont développé les outils informatiques – compilateurs, interpréteurs, assembleurs – qui traduisent ces nouveaux langages en code exécutable efficacement par la machine[18].

29 En 1949 – l'Antiquité de l'histoire de l'informatique – apparaissent les *langages d'assemblage*. Encore très proches de la machine, ces langages remplacent les chiffres par du texte : des mots et des abréviations représentent les instructions du processeur (add pour additionner, cmp pour comparer, etc.) ; des étiquettes nomment les points de programmes et les cases mémoire ; des commentaires en langue naturelle, ignorés lors de l'exécution, permettent d'expliquer et de documenter le programme. L'idée s'impose qu'un programme est destiné non seulement à être exécuté par la machine, mais aussi, et tout autant, à être lu, étudié et modifié par des humains.

30 La Renaissance des langages informatiques débute en 1957 avec le langage Fortran, qui le premier permet d'écrire des expressions arithmétiques proches des notations mathématiques usuelles. Ainsi, les solutions de l'équation $Ax^2 + Bx + C = 0$ se calculent clairement en Fortran :

```
D = SQRT(B*B - 4*A*C)

X1 = (-B + D) / (2*A)

X2 = (-B - D) / (2*A)
```

31 En langage d'assemblage, une douzaine d'instructions seraient nécessaires, au moins une par opération arithmétique élémentaire. C'est le début d'une démarche humaniste où le langage de programmation doit être capable d'exprimer clairement les principaux concepts de son domaine d'utilisation. Ainsi, Fortran, destiné au calcul numérique, accorde une place centrale à la notation algébrique et aux tableaux et matrices. De même, en 1959, Cobol vise l'informatique de gestion et introduit les notions d'enregistrement (*record*) et de fichier structuré, ancêtres de nos bases de données.

32 Les langages Algol et Lisp, apparus tous deux en 1960, marquent un renouveau des idées digne des Lumières. Le langage doit mettre en avant la structure des programmes et faciliter l'écriture et la combinaison de fragments de programmes. Algol et Lisp mettent au premier rang les notions jumelles de *procédure* et de *fonction*. Réutilisables à volonté et dotées d'une interface bien définie, procédures et fonctions sont les briques

de base du logiciel. La récursion, à savoir la possibilité pour une fonction ou une procédure de s'appeler elle-même sur des sous-problèmes, ouvre la voie à une nouvelle pensée algorithmique, au-delà des schémas d'itération usuels. Lisp ouvre une voie supplémentaire, celle du calcul symbolique et non plus exclusivement numérique : un programme Lisp manipule des expressions mathématiques, des formules logiques ou ses propres programmes aussi facilement qu'un programme Fortran manipule des tableaux de nombres.

33 La révolution industrielle balaye le monde du logiciel vers 1965. Le logiciel est de moins en moins produit de manière artisanale par les utilisateurs finaux, de plus en plus par des entreprises spécialisées. L'objectif premier est alors de faire travailler ensemble de grands nombres de programmeurs peu qualifiés, afin de produire des logiciels qui dépassent en taille et en complexité les capacités d'un seul programmeur. Les langages de cette époque, comme PL/I et C, sont essentiellement pragmatiques et se préoccupent moins d'élégance que d'efficacité à utiliser les ressources du matériel. Les besoins croissants en formation donnent naissance à des langages dédiés à l'apprentissage de la programmation, comme Basic, qui permet d'apprendre très rapidement à programmer très mal.

34 En parallèle, les années 1970-1985 voient aussi émerger une contre-culture informatique. Une autre programmation est possible, hors des sentiers battus de l'industrie informatique. Rejetant la domination de la programmation dite « impérative », d'autres paradigmes de programmation émergent et s'incarnent dans des langages d'une originalité radicale : programmation par objets (Simula, Smalltalk), programmation logique (Prolog), programmation fonctionnelle (Scheme, ML, Hope, FP), programmation synchrone (Esterel, Lustre), et d'autres encore.

35 Les années 1980 et 1990 voient cette contre-culture se faire récupérer et devenir la norme. Les anciens hippies vont travailler dans la publicité et la programmation par objets devient le standard de l'industrie, mais *via* des langages (C++, Java) qui dénaturent les idéaux d'origine. La programmation fonctionnelle se répand et gagne en respectabilité suite à des mariages de raison avec la programmation impérative. Le langage OCaml sur lequel j'ai beaucoup travaillé est le résultat d'une telle alliance entre approches fonctionnelles, impératives, et par objets.

36 Dans la même période, on commence à annoncer la fin de la programmation et à remettre en question l'importance du langage. Bientôt, on ne programmera plus : on assemblera des composants logiciels. Certes, il n'est plus nécessaire que chaque programmeur écrive « sa » routine de tri ; mieux vaut la récupérer d'une bibliothèque. Mais assembler lesdits composants logiciels, c'est aussi programmer, à un autre niveau, et avec de nouveaux problèmes. Quelle est l'interface d'un composant ? comment l'utiliser correctement ? quelles garanties fournit-il ? Autant de questions qu'un bon langage de programmation aide à résoudre à travers des mécanismes adaptés à la programmation « à grande échelle » : classes et *packages* (Java, C#...), modules et foncteurs (Standard ML, OCaml), contrats (Eiffel, Racket...), etc.

37 Aujourd'hui, on annonce une autre fin de la programmation : avec l'intelligence artificielle et ses techniques d'apprentissage statistique, le logiciel n'est plus entièrement écrit, mais largement « appris » à partir d'exemples. Une approche à base de langages de programmation peut-elle contribuer à franchir cette nouvelle frontière ? Les travaux récents sur la programmation probabiliste vont dans cette direction[19].

38 Parallèlement à cette évolution des idées et des concepts, la compréhension formelle des langages de programmation – leur *syntaxe* et leur *sémantique* – a beaucoup progressé également, au point qu'on sait aujourd'hui décrire et définir un langage avec la précision des mathématiques. Dès les années 1960 apparaissent des formalismes grammaticaux, souvent inspirés de la linguistique générale, capables non seulement de décrire la syntaxe d'un langage de programmation, mais aussi d'engendrer automatiquement l'analyseur syntaxique correspondant.

39 La sémantique a résisté plus longtemps à la formalisation. Des constructions de programme comme x = x + 1 (incrémenter la variable x) semblent mathématiquement absurdes à première vue. Il faut prendre du recul vis-à-vis de cette étrange syntaxe et introduire une notion explicite d'état du programme – une fonction des variables vers leur valeur courante – pour finalement expliquer cette affectation comme une transformation mathématique de l'état « avant » en l'état « après » son exécution. D'autres idées astucieuses sont nécessaires pour rendre compte d'autres constructions typiques des langages informatiques. Aujourd'hui, nous maîtrisons de nombreuses approches pour donner un sens mathématiquement précis à un programme : sémantiques dénotationnelles, qui interprètent le programme comme un élément d'une structure mathématique (domaines de Scott, jeux à deux joueurs, ou même lambda-calcul de Church) ; sémantiques opérationnelles, qui décrivent les étapes successives des exécutions du programme ; sémantiques axiomatiques, qui décrivent les assertions logiques satisfaites par ces exécutions[20].

40 On peut être surpris qu'il existe des syntaxes et des sémantiques formelles pour des langages de programmation d'usage courant comme C : si le comportement d'un programme C peut être connu avec la précision des mathématiques, comment se fait-il que ce comportement soit si souvent erratique, entre *bugs*, « plantages » et failles de sécurité ? C'est toute la problématique de la vérification formelle du logiciel, que nous abordons maintenant sur un cas très simple : le typage des programmes.

Le typage

41 Dans la vie courante, on n'additionne pas des choux et des carottes ; on ne compare pas des pommes et des oranges ; et on ne mélange pas les torchons et les serviettes. En physique, les équations aux dimensions nous interdisent les mélanges hasardeux : chaque grandeur manipulée a une dimension (durée, distance, masse, etc.), et une équation qui égalise ou additionne deux grandeurs de dimensions différentes est physiquement absurde et donc nécessairement erronée.

42 Généralisant cette sagesse populaire ou physicienne, le typage d'un programme consiste à regrouper les données suivant leur type (entiers, chaînes de caractères, tableaux, fonctions, etc.) et à vérifier que le programme manipule ces données en cohérence avec leurs types. Par exemple, l'expression "hello" (42), qui applique une chaîne de caractères comme si c'était une fonction, est mal typée. Cette vérification simple, qui peut s'effectuer *statiquement*, avant l'exécution du programme, élimine de nombreuses erreurs de programmation et assure un premier niveau de fiabilité du logiciel.

43 Mais l'apport du typage ne s'arrête pas là. Il influence la conception et la structuration des programmes, en permettant aux programmeurs de déclarer les types des structures

de données et des interfaces des composants logiciels, et de faire automatiquement vérifier la cohérence de ces déclarations par un algorithme de typage. Un cran au-dessus, le typage a aussi un impact considérable sur la conception des langages de programmation et sur leur comparaison[21]. Quels traits de langages se prêtent à un typage précis ? Comment bien typer les traits qui nous importent ? Autant de questions qui aident à charpenter le langage. Par exemple, la conception du langage Rust a été guidée par le désir de typer avec précision les structures de données modifiables.

44 Plus profondément encore, les types font apparaître un lien entre langages de programmation et logiques mathématiques : ainsi la correspondance de Curry-Howard, où les types sont vus comme des propositions logiques et les programmes comme des démonstrations constructives. Cette correspondance entre démonstration et programmation a d'abord été observée sur des cas simples par Haskell Curry en 1958 puis par William Howard en 1969. Le résultat semblait tellement anecdotique que Curry le mentionne en quatre pages d'un de ses gros livres[22] et que Howard ne le soumet pas pour publication, se contentant de faire circuler des photocopies de ses notes manuscrites[23]. Rarement photocopie eut un tel impact scientifique, tant cette correspondance de Curry-Howard est entrée en résonance avec le renouveau de la logique et l'explosion de l'informatique théorique des années 1970 pour s'imposer dès 1980 comme un lien structurel profond entre langages et logiques, entre programmation et démonstration. Aujourd'hui, il est naturel de se demander quelle est la signification « logique » de tel ou tel trait de langages de programmation, ou encore quel est le « contenu calculatoire » de tel ou tel théorème mathématique, c'est-à-dire quels algorithmes se cachent dans ses démonstrations. Mon premier cours au Collège de France (celui de l'année 2018-2019), intitulé « Programmer = démontrer ? La correspondance de Curry-Howard aujourd'hui », est consacré à ce bouillonnement d'idées, à la frontière entre logique et informatique.

45 Décidément, cette idée simple d'attacher des types aux données et d'en vérifier le bon usage dans le programme nous a emmenés loin ! Voyons maintenant comment généraliser cette démarche de vérification à bien d'autres propriétés attendues d'un programme, au-delà de la cohérence des types.

La vérification formelle

46 D'un côté, les langages, les outils et les méthodes de programmation ont fait d'immenses progrès. De l'autre, la complexité des logiciels augmente toujours et nous leur confions de plus en plus de responsabilités. Ainsi, le problème central du logiciel se déplace de *programmer* à *convaincre* : convaincre les concepteurs, les développeurs, les utilisateurs, les autorités de régulation ou de certification, voire une cour de justice (si malheur arrive) de la justesse et de l'innocuité du logiciel.

47 Quelles formes prend cette conviction ? Il s'agit de *spécifier* ce que le programme est censé faire (« que voulons-nous ? ») ; *vérifier* que le programme respecte cette spécification (« avons-nous bien écrit le programme ? ») ; et *valider* que le programme et sa spécification sont conformes aux attentes de l'utilisateur (« avons-nous écrit le bon programme ? »). Les spécifications prennent les formes les plus diverses, de la prose à la définition mathématique rigoureuse en passant par des jeux de tests (collection d'exemples de comportements attendus) et des notations semi-formelles (diagrammes, pseudo-code).

48 La méthode de vérification et de validation de loin la plus utilisée est le test, qui consiste à exécuter le programme sur des entrées bien choisies et à examiner les résultats. Il s'agit d'une démarche essentiellement expérimentale, où le logiciel est traité comme un objet de la nature et sa spécification comme une théorie qu'il faut valider expérimentalement. Empiriquement, le test permet d'atteindre de très bons niveaux de fiabilité logicielle, à condition de consacrer beaucoup d'efforts à la construction du jeu de tests. Pour les logiciels critiques, dont dépendent des vies humaines, les niveaux d'exigence sont si élevés que le coût du test devient prohibitif. Scientifiquement, on ne sait attribuer aucune certitude aux résultats du test, à l'exception de rares programmes à états finis qui peuvent se tester exhaustivement. Comme le disait cruellement Edsger Dijkstra en 1969 :

> Testing shows the presence, not the absence of bugs.
> Le test montre la présence, mais pas l'absence d'erreurs.

49 Au-delà du test, les méthodes formelles de vérification établissent, par calcul et déduction logique, des propriétés vraies pour toutes les exécutions possibles du programme[24]. Les propriétés établies vont du bon typage à la robustesse (absence de « plantages » à l'exécution) et jusqu'à la conformité complète du programme à une spécification formelle. Comme nous l'avons déjà remarqué, un programme peut être vu comme une définition mathématique, par l'intermédiaire d'une sémantique formelle pour le langage dans lequel il est écrit. La vérification formelle démontre des propriétés du programme comme autant de théorèmes portant sur cette définition.

50 Les idées et formalismes fondamentaux de la vérification ne datent pas d'hier. Dès 1949, dans une brève communication à un congrès, Turing lui-même suggère d'annoter les programmes avec des assertions logiques (des formules reliant les valeurs des variables)[25]. Cette approche est redéveloppée par Robert W. Floyd[26] en 1967 puis généralisée par C. A. R. Hoare[27] en 1969 sous la forme de *logiques de programmes* qui permettent de raisonner déductivement sur le comportement d'un programme sans avoir à calculer préalablement sa sémantique dénotationnelle ou opérationnelle. Deux techniques de vérification plus spécialisées que les logiques de programmes, et plus faciles à automatiser, apparaissent ensuite : l'interprétation abstraite[28] et la vérification par modèles (*model checking*)[29].

51 Malgré ces avancées conceptuelles, il faut attendre les années 2000 pour que la vérification formelle perce dans l'industrie du logiciel critique, notamment dans le ferroviaire et l'aéronautique. Un des premiers succès, en 1998, est la vérification du logiciel de conduite automatique de la ligne 14 du métro parisien. Raison de ce retard : la vérification formelle d'un programme réaliste nécessite d'énormes quantités de raisonnement déductif et de calcul, bien au-delà des capacités humaines. Il a fallu développer des outils de vérification, automatisant tout ou partie de ces tâches, et faire des progrès considérables dans l'algorithmique de la vérification. C'est un des grands succès de la recherche en informatique des vingt dernières années.

52 Plusieurs familles d'outils sont disponibles, suivant le degré d'automatisation et la précision de la vérification souhaités.

- Un analyseur statique infère automatiquement des propriétés simples d'une variable (son intervalle de variation pour une variable numérique, par exemple) ou de plusieurs (une inégalité linéaire, par exemple). Cela suffit souvent à établir la robustesse du programme.

- Un vérificateur par modèles (*model checker*) peut vérifier automatiquement d'autres types de propriétés, exprimées en logique temporelle, en explorant les états atteignables pendant l'exécution du programme.
- Un vérificateur déductif, aussi appelé « prouveur de programme », est capable de vérifier la conformité complète du programme à sa spécification, pourvu que le programme soit manuellement annoté avec des assertions logiques : préconditions (hypothèses en entrées de fonctions et commandes), postconditions (garanties en sorties) et invariants. L'outil permet alors de vérifier que les préconditions impliquent logiquement les postconditions et que les invariants sont préservés, en utilisant des algorithmes de démonstration automatique.
- Un assistant de preuve comme Coq ou Isabelle permet de mener des raisonnements mathématiques trop complexes pour être automatisés, que ce soit à propos de la sémantique d'un programme ou pour tout autre usage. La démonstration est construite en interaction avec l'utilisateur, mais l'outil en vérifie automatiquement la justesse et l'exhaustivité.

Figure 1

```
bigint[] firstprimes(int n)
{
    assert (n > 0);
    bigint p[] = new bigint[n];
    p[0] = 2;
    bigint m = 3;
  loop:
    for (int i = 1; i < n; m = m + 2) {
        int j = 0;
        while (j < i ∧ p[j] <= √m ) {
            if (m % p[j] == 0) continue loop;
            j = j + 1;
        }
        p[i] = m; i = i + 1;
    }
    return p;
}
```

Calcul des *n* premiers nombres premiers (d'après Knuth, *The Art of Computer Programming*, 1997).

53 Prenons un exemple pour illustrer les capacités de ces outils modernes de vérification. Le programme en figure 1 est un des premiers exemples du traité d'algorithmique de Donald Knuth[30], exprimé dans un langage proche de Java. La fonction firstprimes calcule les n premiers nombres premiers. Elle parcourt les nombres impairs $m = 3, 5, 7, \ldots$ en éliminant ceux qui sont multiples d'un nombre premier déjà trouvé et en conservant les autres.

54 Un outil d'analyse statique par interprétation abstraite peut inférer les inégalités $n > 0$ et $0 \leq i < n$ et $0 \leq j < i$, sans aide supplémentaire du programmeur. Il s'ensuit que les accès p[0], p[i] et p[j] au tableau p ne sortent jamais de ses bornes. L'analyseur a donc montré l'absence d'un *bug* courant et source de nombreuses failles de sécurité.

55 Un outil de vérification déductive peut montrer bien d'autres propriétés de ce code, allant jusqu'à la *correction partielle* : si la fonction s'arrête, elle renvoie bien en résultat le tableau des n premiers nombres premiers, triés par ordre croissant. L'outil n'y arrivera pas tout seul : il faut que le programmeur annote le code avec de nombreuses assertions logiques, notamment des invariants de boucles, qui guident la vérification (fig. 2).

Figure 2

```
loop:
 for (int i = 1; i < m; m = m + 2) {
   /* invariants:
      ∀k, 0 ≤ k < i ⇒ prime(p[k])
      ∀n, 2 ≤ n < m ∧ prime(n) ⇒ ∃k, 0 ≤ k < i ∧ p[k] = n
      ∀j,k, 0 ≤ j < k < i ⇒ p[j] < p[k]
   */
 }
```

L'invariant pour la boucle principale du programme de la figure 1. Il exprime que le sous-tableau p[0] … p[i−1] contient tous les nombres entiers compris entre 2 et m, et rien que ces nombres entiers. De plus, ce sous-tableau est trié par ordre strictement croissant et ne contient donc pas de doublons.

56 La *terminaison* du programme est beaucoup plus difficile à vérifier par des techniques automatiques, car elle est conséquence du fait que l'ensemble des nombres premiers est infini : s'il n'y avait que 10 nombres premiers et qu'on appelle la fonction avec $n = 11$, elle bouclerait à la recherche du 11e nombre premier. On peut ajouter le théorème d'Euclide « il n'y a pas de plus grand nombre premier » comme axiome, mais je ne connais pas de démonstrateur automatique capable de l'utiliser pour montrer la terminaison ; une démonstration interactive avec un assistant comme Coq ou Isabelle semble nécessaire.

57 Un autre exemple d'utilisation de mathématiques non triviales est une amélioration suggérée par Knuth[31], qui consiste à supprimer la condition j < i de la boucle interne, sous prétexte que l'autre condition p[j] <= √m suffit à arrêter cette boucle au bon moment. Démontrer la justesse de cette amélioration est difficile car on a besoin d'un résultat subtil sur la densité des nombres premiers, le postulat de Bertrand, démontré par Pafnouti Tchebychev en 1852, qui montre que pour tout $n > 1$ l'intervalle $]\,n, 2n\,[$ contient au moins un nombre premier. Vérifier ce théorème et l'usage qui en est fait dans le code amélioré nécessite une démonstration interactive en Coq[32]. Cependant, l'outil automatique Why3 est capable de vérifier le code amélioré en prenant en axiome le postulat de Bertrand[33].

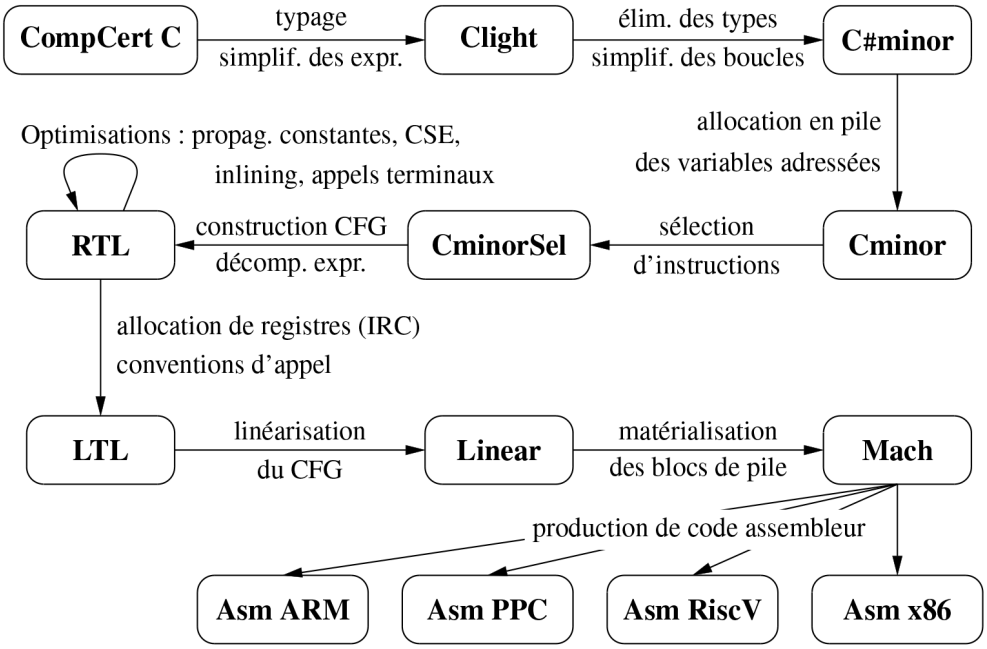
58 On le voit, vérifier la correction totale d'un programme peut nous emmener loin. Mais il n'y a pas de difficultés insurmontables, contrairement à ce que beaucoup de programmeurs ressentent, et moins encore d'impossibilité de principe, contrairement à ce que quelques universitaires ont affirmé[34]. Mais non, le logiciel n'a pas été chassé du paradis mathématique. Aucune divinité n'a décrété « tu programmeras dans la douleur ! avec seulement des tests et quelques diagrammes UML ! »

59 Le principal prérequis à la vérification formelle est l'existence d'une spécification mathématiquement précise des propriétés à vérifier. Il faut avoir mis le problème en équations avant d'espérer le résoudre ! Voilà qui ne surprend pas les physiciens, mais n'est pas toujours compris des programmeurs et encore moins des décideurs. Certains domaines d'application savent depuis longtemps mettre les problèmes en équation : c'est le cas pour les lois de contrôle-commande qui sont au cœur de nombreux logiciels critiques, comme la régulation des trains ou les commandes électroniques de vol des avions. D'autres domaines ont récemment fait d'énormes progrès vers la spécification et la vérification formelles : protocoles et applications cryptographiques[35], composants de systèmes d'exploitation[36]. À l'autre extrémité du spectre, l'apprentissage statistique, qui est au cœur de l'intelligence artificielle contemporaine, est souvent appliqué à des tâches de perception pour lesquelles aucune spécification précise n'existe : comment caractériser ce qu'est une « bonne » reconnaissance de la parole ou une « bonne » classification d'images plus précisément que par un taux de réussite sur un jeu d'exemples ? Mais dans les cas où une spécification précise est disponible, un réseau de neurones profond peut tout à fait être vérifié formellement une fois la phase d'apprentissage terminée, comme le montrent les travaux de Guy Katz et de ses collègues sur le système ACAS-Xu d'évitement de collisions aériennes[37].

60 Un autre obstacle sur la voie du logiciel formellement vérifié est la confiance que l'on peut accorder aux outils informatiques qui participent à sa vérification et à sa production. Ces outils sont des programmes comme les autres, potentiellement erronés. Une erreur de conception ou d'implémentation dans un analyseur statique ou dans tout autre outil de vérification peut lui faire ignorer un chemin d'exécution dangereux ou une cause d'erreur dans le programme analysé, et donc lui faire conclure à tort que le programme analysé est correct. De plus, les outils de vérification opèrent rarement sur le code machine qui s'exécute réellement, mais plutôt sur le code source, écrit en langage de haut niveau, voire sur un modèle de l'application. Un second risque, plus insidieux, apparaît alors : les compilateurs et générateurs automatiques de code, qui traduisent le code source ou le modèle en code machine exécutable, pourraient faire des contresens et produire un exécutable incorrect à partir d'un code source ou d'un modèle formellement vérifiés.

61 *Quis custodiet ipsos custodes* ? Si les outils de vérification et de compilation sont les gardiens de la qualité du logiciel, qui gardera ces gardiens ? Le test n'est pas très efficace, tant ces outils sont des programmes complexes mettant en œuvre des algorithmes subtils de calcul symbolique. En revanche, ces programmes ont des spécifications assez simples en termes de sémantique des langages concernés, ce qui ouvre la voie à une vérification formelle. Mes travaux récents, et ceux de nombreux collègues, explorent cette idée de vérifier formellement les outils qui participent à la vérification et à la compilation des logiciels critiques. En utilisant l'assistant de preuve Coq, nous avons développé et vérifié CompCert[38], un compilateur réaliste pour le langage C (voir fig. 3). Les bonnes performances de CompCert et la difficulté de sa vérification en font une « première » dans le domaine. Deux autres projets réutilisent les mêmes approches pour vérifier d'autres outils : Verasco[39], un analyseur statique par interprétation abstraite, et Velus[40], un générateur de code pour le langage réactif Lustre.

Figure 3



Partie formellement vérifiée du compilateur CompCert. Elle traduit le sous-ensemble CompCert du langage C en langage d'assemblage pour les 4 architectures cibles, en passant par 8 langages intermédiaires. La traduction se compose de 16 passes, dont certaines sont des « optimisations » qui éliminent des inefficacités et améliorent les performances du code produit. Une telle architecture est standard pour un compilateur moderne. La nouveauté de CompCert est que chaque langage (source, intermédiaire, cible) est doté d'une sémantique formelle, et que chaque passe de compilation est vérifiée comme préservant ces sémantiques, grâce à l'assistant de preuve Coq.

Conclusion

62 Arrivé à la fin de cet exposé, la première chose que je voudrais souligner, c'est l'ampleur des progrès dans le monde du logiciel depuis l'apparition de l'ordinateur. Langages de programmation de haut niveau, compilateurs, sémantiques, systèmes de types, spécifications formelles, logiques de programmes, outils automatisant entièrement ou partiellement la vérification, vérification de ces outils ainsi que des compilateurs : autant d'étapes franchies au cours des soixante dernières années qui décuplent – voire multiplient par mille – nos capacités à créer du logiciel sûr et sécurisé.

63 Allons-nous atteindre la perfection logicielle, cet idéal où le logiciel se comporte exactement comme prescrit par sa spécification, et où la programmation devient invisible ? Comme tout idéal, il s'éloigne à mesure que nous nous en approchons…

64 La bien mal nommée « intelligence artificielle » réalise des tâches, de perception notamment, inimaginables il y a quelques années encore, mais fait apparaître des « boîtes noires » vulnérables aux biais, au bruit, et à la malveillance[41]. En termes de fiabilité du logiciel, cela nous ramène vingt ans en arrière et nécessite de nouvelles méthodes de vérification et de validation.

65 Le matériel se révèle moins infaillible que les auteurs de logiciels le supposaient. Les vulnérabilités de type « Spectre » montrent qu'il a décidément bien du mal à garder un secret[42].

66 Les méthodes formelles restent difficiles à mettre en place et continuent à terrifier beaucoup de programmeurs. Un premier pas serait de rendre plus agréable l'écriture de spécifications, par exemple *via* de nouveaux langages de spécification.

67 Enfin, il faut s'interroger sur la manière dont nous enseignons l'informatique et même les mathématiques. Nos étudiants confondent souvent les quantificateurs « pour tout » et « il existe ». Difficile d'écrire des spécifications sans maîtriser les notions de base de la logique…

68 La logique ! On y revient encore et toujours ! C'est le *leitmotiv* de cette leçon : l'informatique fondamentale qui naît de l'*hubris* des logiciens du début du XX^e^ siècle ; les langages de programmation et leurs sémantiques ; les spécifications et les logiques de programmes ; jusqu'aux systèmes de types qui suggèrent que programmer et démontrer, c'est la même chose… Finalement, le logiciel serait-il juste de la logique qui s'exécute ? Pas toujours (l'erreur est humaine !) ; pas seulement (il y a d'autres dimensions à considérer) ; mais si seulement ! Être l'incarnation de la logique est une des meilleures choses que l'on puisse souhaiter au logiciel. Au moment où nous confions de plus en plus de responsabilités aux logiciels et déléguons de plus en plus de décisions à des algorithmes, dans l'espoir naïf qu'ils feront moins d'erreurs que les humains, ou pour d'autres raisons moins avouables[43], nous avons plus que jamais besoin de rigueur mathématique pour exprimer ce qu'un programme doit faire, raisonner sur ce qu'il fait, et maîtriser les risques qu'il présente. C'est dans cet équilibre entre rigueur formelle et créativité débridée que s'inscrit le logiciel ; c'est à nous, informaticiens et citoyens, de construire cet équilibre.

NOTES

1. M. Moneghetti, « Collège de France : 40 leçons inaugurales », https://www.franceculture.fr/emissions/college-de-france-40-lecons-inaugurales, émission diffusée en juillet et août 2016 (consulté le 4 février 2019).

2. P. Mounier-Kuhn, *L'Informatique en France de la Seconde Guerre mondiale au Plan Calcul. L'émergence d'une science*, Paris, Presses universitaires Paris-Sorbonne, 2010.

3. E. Lazard et P. Mounier-Kuhn, *Histoire illustrée de l'informatique*, Les Ulis, EDP Sciences, 2016.

4. *Ibid.*, p. 43-45.

5. L. F. Menabrea et A. A. Lovelace, *Sketch of the analytical engine invented by Charles Babbage*, 1842 ; en ligne : Fourmilab Switzerland, http://www.fourmilab.ch/babbage/sketch.html (consulté le 18 février 2019).

6. E. Lazard et P. Mounier-Kuhn, *Histoire illustrée de l'informatique*, *op. cit.*, chap. 4 et 5.

7. G. Berry, *L'Hyperpuissance de l'informatique*, Paris, Odile Jacob, 2017.

8. G. Dowek, *Les Métamorphoses du calcul. Une étonnante histoire de mathématiques*, Paris, Le Pommier, 2007.

9. G. W. Leibniz, *Nova Methodus pro Maximis et Minimis*, *in* : *Acta Eruditorum*, oct. 1684.

10. M. Davis, *The Universal Computer: The Road from Leibniz to Turing*, Boca Raton, CRC Press, 2012, chap. 1.

11. *Ibid.*, chap. 2-4.

12. G. Dowek, *Les Métamorphoses du calcul*, *op. cit.*, chap. 4.

13. Composé d'axiomes (par exemple « $n + 0 = n$ pour tout n ») et de règles de déduction (par exemple le *modus ponens*, « de $P \Rightarrow Q$ et de P je peux déduire Q »).

14. M. Davis, *The Universal Computer*, *op. cit.*, chap. 5.

15. K. Gödel, « Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme », I. *Monatshefte für Mathematik und Physik*, vol. 38, n° 1, 1931, p. 173-198 ; traduit en anglais dans J. van Heijenoort, *From Frege to Gödel: A Source Book in Mathematical Logic, 1879-1931*, Cambridge (Mass.), Harvard University Press, 1977.

16. A. Church, « A note on the Entscheidungsproblem », *Journal of Symbolic Logic*, vol. 1, n° 1, 1936, p. 40-41 ; A. M. Turing, « On computable numbers, with an application to the *Entscheidungsproblem* », *Proceedings of the London Mathematical Society*, s. 2, vol. 42, n° 1, 1937, p. 230-265.

17. I. L. Markov, « Limits on fundamental limits to computation », *Nature*, vol. 512, 2014, p. 147-154.

18. P. van Roy et S. Haridi, *Programmation. Concepts, techniques et modèles*, Paris, Dunod, 2007.

19. J. van de Meent *et al.*, *An Introduction to Probabilistic Programming, Computing Research Repository*, 2018 : https://arxiv.org/abs/1809.10756.

20. H. R. Nielson et F. Nielson, *Semantics with Applications: An Appetizer*, Londres, Springer, 2007.

21. B. C. Pierce, *Types and Programming Languages*, Cambridge (Mass.), MIT Press, 2002.

22. H. B. Curry et R. Feys, *Combinatory Logic*, Amsterdam, North-Holland, 1958.

23. W. A. Howard, *The Formulae-as-types Notion of Construction*, 1969 ; reproduit dans J. P. Seldin et J. R. Hindley (dir.), *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, New York, Academic Press, 1980, p. 479-490.

24. J.-F. Monin, *Introduction aux méthodes formelles*, Paris, Hermès, 2000.

25. Reproduit et commenté dans L. Morris et C. B. Jones, « An early program proof by Alan Turing », *Annals of the History of Computing*, vol. 6, n° 2, 1984, p. 129-143.

26. R. W. Floyd, « Assigning meaning to programs », *in* J. T. Schwartz (dir.), *Mathematical Aspects of Computer Science* (*Proceedings of Symposia on Applied Mathematics*, vol. 19), New York, American Mathematical Society, 1967, p. 19-32.

27. C. A. R. Hoare, « An axiomatic basis for computer programming », *Communications of the ACM*, vol. 12, n° 10, 1969, p. 576-585.

28. P. Cousot et R. Cousot, « Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints », *POPL '77: Conference Record of the Fourth ACM Symposium on Principles of Programming Languages*, 1977, p. 238-252.

29. E. A. Emerson et E. M. Clarke, « Characterizing correctness properties of parallel programs using fixpoints », *in* J. W. de Bakker et J. van Leeuwen (dir.), *Automata, Languages and Programming, 7th Colloquium* (*Lecture Notes in Computer Science*, vol. 85),

Berlin/Heidelberg/New York, Springer, 1980, p. 169-181 ; J. Queille et J. Sifakis, « Specification and verification of concurrent systems in CESAR », *in* M. Dezani-Ciancaglini et U. Montanari (dir.), *International Symposium on Programming, 5th Colloquium* (*Lecture Notes in Computer Science*, vol. 137), Berlin/Heidelberg, Springer, 1982, p. 337-351.

30. D. E. Knuth, *The Art of Computer Programming*, vol. 1 : *Fundamental Algorithms*, 3e éd., Boston, Addison Wesley, 1997, section 1.3.2, programme P.

31. *Ibid*., section 1.3.2, exercice 6.

32. L. Théry, « Proving pearl: Knuth's algorithm for prime numbers », *in* D. Basin et B. Wolff (dir.), *Theorem Proving in Higher Order Logics: 16th International Conference, TPHOLs 200* (*Lecture Notes in Computer Science*, vol. 2758), Berlin/Heidelberg, Springer, 2003, p. 304-318.

33. J.-C. Filliâtre, *Knuth's prime numbers*, *in* : *Gallery of verified programs*, http://toccata.lri.fr/gallery/knuth_prime_numbers (consulté le 4 février 2019).

34. Pour une analyse de ces prétendues impossibilités, voir A. Asperti, H. Geuvers et R. Natarajan, « Social processes, program verification and all that », *Mathematical Structures in Computer Science*, vol. 19, n° 5, 2009, p. 877-896.

35. V. Cortier et S. Kremer, « Formal models and techniques for analyzing security protocols: A tutorial », *Foundations and Trends in Programming Languages*, vol. 1, n° 3, 2014, p. 151-267.

36. G. Klein *et al.*, « seL4: Formal verification of an operating-system kernel », *Communications of the ACM*, vol. 53, n° 6, 2010, p. 107-115.

37. G. Katz *et al.*, « Reluplex: An efficient SMT solver for verifying deep neural networks », *in* R. Majumdar et V. Kunčak, *Computer Aided Verification: 29th International Conference, CAV 2017* (*Lecture Notes in Computer Science*, vol. 10426), Berlin/Heidelberg, Springer, 2017, p. 97-117.

38. X. Leroy, « Formal verification of a realistic compiler », *Communications of the ACM*, vol. 52, n° 7, 2009, p. 107-115.

39. J. Jourdan *et al.*, « A formally-verified C static analyzer », *POPL 2015: Proceedings of the 42nd Symposium on Principles of Programming Languages*, 2015, p. 247-259.

40. T. Bourke *et al.*, « A formally verified compiler for Lustre », *PLDI 2017: Proceedings of the 38th Conference on Programming Language Design and Implementation*, 2017, p. 586-601.

41. C. O'Neil, *Algorithmes, la bombe à retardement*, Paris, Les Arènes, 2018.

42. C. Maurice, « Après Meltdown et Spectre, comment sécuriser les processeurs ? », *Journal du CNRS*, mars 2018.

43. S. Abiteboul et G. Dowek, *Le Temps des algorithmes*, Paris, Le Pommier, 2017.

AUTEUR

XAVIER LEROY

Professeur au Collège de France, titulaire de la chaire Sciences du logiciel