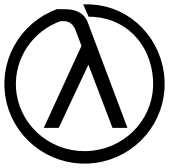

L’histoire de la tragique domination des langages impératifs

Il existe divers paradigmes de programmation qui partagent les langages de programmation en deux grandes familles : les langages impératifs et les langages déclaratifs.

Même si les plus modernes d’entre eux s’imprègnent de concepts issus de la programmation déclarative (en particulier fonctionnelle), il demeure que l’écrasante majorité des langages de programmation qui constituent le paysage de l’industrie mondiale du logiciel sont des langages impératifs. Et c’est une situation qui dure depuis les débuts de l’informatique moderne[1].

À une innovation près, si elle était survenue juste un peu plus tôt dans cette Histoire, il aurait pourtant pu en être autrement ; pour le mieux.

1. L’histoire de cette domination

Les langages impératifs dominent donc, qui plus est de façon écrasante, la scène du développement logiciel.

Cela tient en partie au fait que l’architecture des ordinateurs, en particulier de leurs processeurs, appelle directement une approche impérative. Il était donc tout naturel que les premiers langages de développement suivent cette logique. L’informatique était alors une science naissante et les limites techniques des machines d’alors imposaient que les programmeurs s’en accommodent. L’idée que ce serait idéalement plutôt à la machine de s’adapter à l’Homme faisant alors figure d’objectif inatteignable.

Mais assez vite il a été possible de théoriser d’autres approches du raisonnement et de la programmation que l’approche strictement impérative, car si l’informatique est une discipline jeune (au regard de l’Histoire des sciences) il ne s’agit jamais que de mathématiques appliquées. Or les mathématiques sont une discipline incomparablement plus ancienne et plus mûre.

Ainsi, dès 1958, on a vu apparaître le premier langage fonctionnel (impur) : Lisp. C’est en 1958 également que John McCarthy proposait déjà d’utiliser la logique comme langage déclaratif de représentation des connaissances. Les travaux qui s’ensuivirent aboutirent à la création du langage Prolog, **le** langage de programmation logique, en 1972.

Programmation **fonctionnelle** et programmation **logique** sont les deux grandes familles de la programmation déclarative.

---

1. On traitera ici d’informatique « moderne » en faisant commencer son Histoire dans les années 1940, avec les travaux d’Alan Turing pendant la seconde guerre mondiale et en laissant de côté les expériences antérieures intéressantes mais anecdotiques de mécano-informatique, comme la fameuse machine Pascaline.

Si la programmation logique s'avère plus spécifiquement adaptée à certains types de problèmes que sont les systèmes experts et certaines formes précoces d'intelligence artificielle, la programmation fonctionnelle a en revanche le potentiel de permettre la réalisation de programmes dans un ensemble de domaines très vaste, qui est le même que celui des langages impératifs, c'est à dire tout ce que peut exprimer l'algorithmique. Les deux approches peuvent donc être considérées comme étant en concurrence directe.

Il peut être prouvé que tout algorithme (ayant à traiter des données organisées en un ensemble pouvant être parcourru) peut être décliné en une version itérative et en une version récursive. Les langages impératifs permettent d'exprimer aussi bien les formes itératives que les formes récursives de ces algorithmes. Les langages fonctionnels sont quant à eux essentiellement cantonnés aux formes récursives.

Cette limitation des langages fonctionnels ne permet cependant pas directement d'expliquer la préférence de l'industrie pour les langages impératifs. Elle n'est en effet pas rédibitoire, car du fait que tout problème peut être aussi bien résolu par l'une ou l'autre des approches, les langages qui ne permettent que l'approche récursives permettent néanmoins de résoudre tous les problèmes. Qui plus est, l'approche récursive est souvent plus élégante et plus proche du raisonnement humain, donc plus intelligible par les programmeurs. Il y aurait donc des raisons de la préférer à l'approche itérative et de se consoler de l'absence de cette dernière.

Mais, à cette ère de l'informatique avec ses ordinateurs aux capacités encore extrêmement limitées, les traitements récursifs s'avèrent bien plus coûteux en mémoire que leurs contreparties itératives. Les ordinateurs utilisent en effet dans leur mémoire une pile de traitement et chaque récursion d'une fonction nécessite l'empilement d'un contexte mémoire complet pour cette fonction ; là où la version itérative du traitement stockera des données mutables dans une autre portion de mémoire (le tas) dont le volume n'augmentera pas proportionnellement au nombre d'itérations à réaliser.

2. Ce qui aurait pu tout changer

Telle était la situation dans le début des années 1970, époque à laquelle l'informatique a connu une accélération de son développement économique qui lui a fait prendre une dimension que l'on pouvait commencer à qualifier d'industrielle. Un nombre croissant d'entreprises s'apprêtait à développer du logiciel et il leur fallait prendre des décisions quant aux technologies qui seraient employées pour ce faire, au premier rang desquelles des langages de programmation, et y former du personnel à grands frais.

Au moment où ces décisions ont du être prises, l'affaire semblait entendue. Pour intéressante que la programmation fonctionnelle puisse paraître, l'avantage économique des langages impératifs s'avérait irrésistible et donc décisif.

Pourtant, la technique a fini par remédier au handicap de la récursion sur l'itération. Une innovation appelée optimisation de la récursion terminale a en effet finalement permis d'élever la récursion au même niveau d'efficacité en mémoire que l'itération.

Qui plus est, cette optimisation s'avère automatisable dans les compilateurs et interpréteurs des langages fonctionnels et non dans ceux dédiés aux langages impératifs.

Cette innovation aurait donc pu tout changer… si elle était survenue plus tôt. Mais au moment où elle est arrivée, les entreprises du logiciels (pour peu qu'elles en aient eu vent) ont du considérer que trop de budget avait déjà été investi dans les langages impératifs pour revoir leurs copies.

Cette inertie scellera le destin de l'industrie informatique pour les décennies à venir.

3. Pourquoi cette domination est tragique

La branche impérative de la famille des langages de programmation a connu son lot d'innovations successives. À la programmation monolithique a succédé la programmation structurée, la programmation procédurale et finalement la programmation orientée objet.

La programmation orientée objet est donc l'aboutissement de la programmation impérative. Et il est intéressant de noter que si les premières transformations ayant occasionné le passage à la programmation structurée, puis à la programmation procédurale, ont largement concerné la façon définir les traitements, il l'est tout autant de noter que la dernière, le passage à la programmation orientée objet, n'a rien apporté de très fondamental en la matière et a surtout concerné l'agencement des données.

Il est également à noter que, si vu de l'extérieur le développement de l'informatique semble un succès retentissant au regard de la place qu'elle a prise dans notre société moderne, vu de l'intérieur le portrait est nettement moins reluisant. Nous autres programmeurs croulons sous des déferlantes de bugs à corriger et 70% des projets logiciels lancés dans le monde se soldent en fait par des échecs.

Or, les difficultés que nous éprouvons sont incomparablement plus souvent liées à la question de l'agencement des traitements qu'à celle de l'agencement des données. On se doit également de noter que les améliorations successives des langages impératifs, celles là même qui ont abouti à la programmation orientée objet qui est aujourd'hui considérée comme l'état de l'art de l'industrie informatique, ne sont pas parvenues à résoudre ce problème. La programmation impérative semble être arrivée au bout de ce qu'elle avait à proposer en matière d'agencement des traitements… et cela se révèle insuffisant.

De son côté, la programmation fonctionnelle, parce qu'elle est bâtie sur des expressions qui définissent des intentions (le quoi) plutôt que des méthodes (le comment), permet des automatisations et des optimisations sur les traitements qui sont fondamentalement hors de portée, par définition même, de la programmation impérative. La limitation des effets de bord à des formes explicites (encadrées par des monades), l'agencement des données s'appuyant sur la rigueur de la théorie mathématique des types ou encore la mémoisation automatique des résultats que permet la transparence référentielle des « vraies » fonctions sont autant d'exemples d'apports qui éliminent des catégories entières de bugs que l'on retrouve malheureusement régulièrement lorsque l'on pratique la programmation impérative.

Les pistes les plus prometteuses en matière d'innovations propres à résoudre le problème général de marée haute de bugs que subit l'industrie du logiciel semblent donc résider dans la programmation déclarative, et plus particulièrement dans la programmation fonctionnelle. C'est en cela que le virage jadis pris en faveur de la programmation impérative, du fait de l'arrivée trop tardive d'une innovation qui aurait

pu tout changer, apparaît aujourd'hui comme tragique à ceux qui ont connaissance de cet enchaînement d'évènements et conscience de ses conséquences.

Les concepteurs des langages impératifs les plus utilisés, notamment Java, Javascript et C#, ont pris la mesure de cet état de fait. Aussi ces langages orientés objet ont-ils subits des réformes pour intégrer des syntaxes inspirées de la programmation fonctionnelle.

Mais ils demeurent malgré cela fondamentalement des langages impératifs. Par construction, ils ne peuvent pas procurer les garanties que peuvent procurer les langages de programmation fonctionnelle pure comme Haskell. Ils ne sont pas même devenus des langages de programmation fonctionnelle impure comme ML ou comme Scala. Ce sont juste des langages impératifs saupoudrés de syntaxe fonctionnelle. Cela améliore leur puissance d'expression, mais pas leur fiabilité générale.