

Métaprogrammation et libre disponibilité des sources

deux défis informatiques d'aujourd'hui*

François-René Rideau Đặng-Vũ Bân
francoisrene.rideau@cnet.francetelecom.fr
<http://fare.tunes.org/>

CNET DTL/ASR (France Telecom) †
38-40 rue du general Leclerc
92794 Issy Moulineaux Cedex 9, FRANCE

Résumé

Nous présentons de façon complètement informelle la métaprogrammation, dont nous esquissons une théorie. Nous expliquons en quoi elle représente un enjeu majeur pour l'informatique d'aujourd'hui, dès lors que l'on examine les processus sous-jacents au développement logiciel. Nous montrons par les mêmes considérations, en quoi la métaprogrammation est liée à un autre défi de l'informatique, la libre disponibilité des sources des logiciels, et comment ces deux phénomènes se complètent naturellement.

1 Introduction

Dans les conditions primitives des débuts de l'informatique, la capacité des machines était si petite qu'un seul homme pouvait embrasser de la pensée l'ensemble du fonctionnement d'un programme jusqu'à ses moindres détails. Cependant, les facultés humaines n'ont pas changé depuis, tandis que la capacité des machines a suivi une progression géométrique soutenue. Pour profiter de l'évolution technologique, il a donc été nécessaire de développer et d'utiliser des outils conceptuels et des langages de programmation de plus en plus abstraits.

Mais quels que soient les progrès effectués individuellement par les programmeurs, il est un niveau de complexité depuis fort longtemps dépassé au-delà duquel nul ne peut seul concevoir dans son intégralité un programme qui tire pleinement parti des machines existantes. Il est donc primordial de développer des méthodes d'échange, de coopération, d'accumulation, de construction, pour permettre l'élaboration de programmes complexes ; tel est le domaine du génie logiciel [Brooks, 1995].

Or, au centre de tout processus de construction logicielle, il y a la manipulation des sources des programmes. Améliorer ces processus, faciliter la tâche du programmeur, c'est délivrer le programmeur de toutes les opérations répétitives et conceptuellement redondantes, pour qu'il puisse se concentrer sur

*Cet essai, publié lors de la conférence «Autour du Libre 1999» organisée à l'ENST-Bretagne, est disponible sous la licence GNU GPL, version 2 ou ultérieure.

†Les opinions exprimées dans cet essai n'engagent que leur auteur, et ne sauraient en aucun cas être tenues comme une position officielle du CNET ou de France Telecom.

l'essentiel de la programmation, c'est-à-dire sur les problèmes qui n'ont encore jamais été résolus. C'est donc automatiser autant que possible la manipulation du code, en faisant exécuter de manière fiable par la machine toutes les tâches subalternes qui ne posent plus de problème théorique. Et automatiser la programmation, c'est par définition *métaprogrammer*.

La métaprogrammation, art de programmer des programmes qui lisent, manipulent, ou écrivent d'autres programmes, apparaît donc naturellement dans la chaîne de développement logiciel, où elle joue un rôle essentiel, ne fût-ce «que» sous la forme de compilateurs, interpréteurs, débogueurs. Cependant, elle n'est quasi jamais intégrée consciemment dans les processus de développement, et la prise de conscience de son rôle est précisément le premier pas vers un progrès en la matière.

Maintenant, même si nous voulions nous consacrer à la tâche technique, suffisamment ardue en elle-même, consistant à explorer les méthodes d'automatisation du processus de développement logiciel, il nous est impossible d'ignorer la précondition nécessaire à l'utilisation de toute telle méthode ainsi que de tout travail incrémental ou coopératif : la disponibilité des sources.

Cette disponibilité est moins que jamais un problème technique, grâce à l'avènement des télécommunications numériques ; mais elle est plus que jamais un problème politique, à l'heure où la quasi-totalité de l'information sur la planète est sous le contrôle de monopoles éditoriaux organisés en puissants groupes de pression. C'est pour la libre disponibilité des sources, voire de l'information en général, que milite le mouvement pour le Libre Logiciel, qui combat les barrières artificielles que sont les privilèges légaux de «propriété intellectuelle».

Métaprogrammation et libre disponibilité des sources, tels sont les deux défis majeurs, intimement liés, auxquels a à faire face l'informatique aujourd'hui, et qui acquièrent une importance chaque jour plus grande. Tous deux trouvent la même justification cybernétique³ dans la nécessité d'adapter le processus de développement logiciel à des programmes de plus en plus élaborés concernant un public de plus en plus étendu, où chacun ne peut apporter qu'une petite pierre. Telle est du moins notre conviction, que nous allons tenter de vous faire partager.

2 Métaprogrammation

2.1 la métaprogrammation dans le quotidien

Il est possible de voir l'impact de la métaprogrammation sur les processus traditionnels de développement, en catégorisant les programmes existants comme processus transformationnels avec entrées et sorties, en nous intéressant spécifiquement à celles parmi ces entrées et sorties qui sont (avec une certaine dose d'arbitraire) considérées comme du «code», par opposition à des «données». On notera $n \rightarrow p$ la catégorie des programmes ayant n entrées constituées par du «code», et p sorties constituées par du «code». Voici alors quelques exemples de programmes rentrant dans les plus simples de ces catégories grossières.

- $0 \rightarrow 0$: ne prend ni ne renvoie de code.
tout programme «normal», de base, non méta.
- $1 \rightarrow 0$: prend du code en entrée, n'en renvoie pas en sortie.
Interpréteur : prend un programme et l'exécute (un shell, un interpréteur de scripts, une boucle d'interaction Lisp ; les deux derniers cas suggèrent que d'une manière interne, les interpréteurs efficaces feront appel à un compilateur). *Inspecteur* de code : prend un programme, et extrait certaines données (documentation,

³ Le mot «cybernétique» est construit sur la racine grecque *κυβερνήτης*, qui désigne le pilote d'un navire, et désignait déjà au sens figuré *gouverneur*, mot qui en dérive étymologiquement. La cybernétique [Wiener, 1957] est la science du contrôle, c'est-à-dire du flux d'information et de la décision.

graphes de flux d'information, statistiques sur la productivité, etc.). *Testeur* automatique: teste automatiquement le comportement ou mesure les performances d'un programme dans diverses situations. *Analyseur* de code: étudie le comportement du programme, recherche certaines défaillances, vérifie que le programme est correct vis-à-vis de certaines spécifications ou de certains critères.

- $0 \rightarrow 1$: produit du code en sortie sans en prendre en entrée.
Précompilateur de données: adapte et inclut des données externes dans le code (images, sons, fontes, tables de gestion clavier, etc.); précalcule des tables de trigonométrie (sinus/cosinus pour TFR), de chiffrement ou de déchiffrement (CRC, DES), produit des routines optimisées pour l'affichage d'icônes, génère des archives auto-extractibles, etc. *Générateur* automatique de programme: produit un programme optimisé à la résolution d'un problème particulier dans une classe connue.
- $1 \rightarrow 1$: prend un programme en entrée, écrit du programme en sortie.
Traducteur automatique (d'un langage à un autre), en particulier *Compilateur* (d'un langage de «haut niveau» vers un langage de «bas niveau»). *Optimiseur* ou *Évaluateur Partiel*: adapte le code pour de meilleures performances dans son environnement futur d'exécution. *Extracteur* de code secondaire: produit automatiquement du code pour l'exportation de routines, l'initialisation et la finalisation, la gestion d'erreur, etc. *Instrumenteur*: insère du code supplémentaire pour l'interface ou le débogage. *Compacteur*: produit du code auto-décompactant. *Stylisateur*: rend le code plus facile (ou plus difficile) à lire et à comprendre. *Dérivateur*: calcule la dérivée de la fonction calculée par le programme en entrée. *Solveur*: cherche un programme solution d'une spécification donnée.
- $2 \rightarrow 0$: prend deux programmes en entrée, ne renvoie aucun programme.
Métainterpréteur: interprète le second programme selon le langage décrit par le premier. *Vérificateur*: prend une spécification (une proposition) et un programme (une preuve), et vérifie que le second est conforme au premier (l'établit). *Analyseur différentiel*: compare deux programmes, tente de trouver des différences, de repérer un cas où l'un n'implémente pas l'autre, de trouver un ensemble de modifications le plus court possible pour passer d'un programme à l'autre, etc. *Coexécuteur*: exécute les deux programmes en même temps (ou quelque combinaison que ce soit), par exemple pour profiter selon les cas, de celui qui est le plus efficace, pour les faire jouer l'un contre l'autre, ou simplement pour permettre à l'utilisateur de tirer partie des capacités et de l'un, et de l'autre. *Explicateur*: prend un programme et une exécution du programme ayant mené à un résultat incongru, et tente de fournir une explication aussi pertinente que possible du dysfonctionnement réel ou apparent du programme (des incarnations très primitives de tels programmes sont répandues sous le nom de débogueurs).
- $2 \rightarrow 1$: prend deux programmes en entrée, en renvoie un.
Arpenteur de code: prend un métaprogramme et un programme, et parcourt le second selon les instructions spécifiées par le premier. *Métacompilateur*: prend la spécification d'un langage et un programme dans ce langage, et produit un programme qui implémente ce dernier dans un troisième langage (par exemple en langage d'assemblage). *Cooptimiseur*: combine deux programmes en un seul qui les coexécute efficacement. *Tresseur d'aspects*: prend plusieurs aspects d'un même programme putatif, et produit un programme qui implémente à la fois ces multiples aspects. *Générateur de mise à jour*: prend en entrée un programme et sa modification, et génère automatiquement des convertisseurs de données de l'ancien format vers la nouvelle version, ou un programme de «patch» transformant l'ancien programme dans le nouveau, ou un adaptateur permettant aux utilisateurs d'une version de faire tourner leurs applications avec l'autre, etc.
- $1 \rightarrow 2$: prend un programme en entrée, en renvoie deux.
Séparateur de phases: prend un programme, et le sépare en deux, typiquement, un exécutif, partie «effective» destinée à tourner dans l'environnement cible, et un adaptateur, partie d'interface entre l'environnement cible et un environnement hôte (logiciel embarqué et logiciel de contrôle, routines distantes et routines locales dans un système réparti, travail d'un coprocesseur spécialisé et d'un processeur générique, logiciel utilisateur et installateur automatisé, bibliothèque de routines et générateur de solutions optimisées par dessus cette bibliothèque, etc.).

On pourrait continuer vers des exemples arbitrairement élaborés, ou simplement combiner les exemples précédents. On pourrait aussi raffiner ce système de type, pour regarder, par-delà la simple arité des programmes, le type des arguments et résultats (et ainsi de suite, récursivement), avec des critères plus précis qu'être ou non un «programme» (avec pour chaque programme, le langage dans lequel il est écrit), etc. Ces développements sont laissés en tant qu'exercice pour le lecteur.

En regardant la liste d'exemples ci-dessus, on ne peut que constater qu'il existe dans chacune de ces catégories des logiciels qui sont effectivement utilisés quotidiennement, à plus ou moins grande échelle, par des programmeurs du monde entier. La métaprogrammation existe donc déjà, et fait preuve chaque jour de son utilité.

Cependant, par l'inspection de la même liste, on constate aussi que la plupart de ces métaprogrammes sont écrits de manière *ad-hoc*, et *a posteriori*, au gré des besoins immédiats, sans aucune infrastructure générale pour les développer (quoiqu'avec une collection d'outils décousue), et pis encore, sans aucun cadre pour les penser. Certes, la métaprogrammation est présente, mais rares sont ceux qui en ont conscience, et plus rares encore ceux qui l'utilisent activement [Pitrat, 1990]; la tradition informatique semble plus encline à créer un cas particulier pour chacun des concepts présentés qu'à les rapprocher par une théorie utile et systématique.

2.2 esquisse d'une théorie de la métaprogrammation.

Il n'est pas particulièrement compliqué de formaliser l'idée de métaprogrammation, une fois qu'on s'y attache; en effet, toutes les idées sous-jacentes existent déjà, et n'attendent que d'être collationnées en un tout cohérent. Ci-dessous est exposé un court résumé d'une telle théorie, accessible aux personnes ayant une culture de base en informatique théorique ou en logique mathématique. Nous invitons les autres lecteurs à continuer leur lecture en sautant sans scrupule les paragraphes trop difficiles pour eux. Quant à ceux qui seraient intéressés par davantage de détails techniques, nous leur préparons un autre article, plus formel.

Il s'agit d'abord de définir la notion abstraite de *langage* informatique. Un examen des propriétés que l'on en attend permet de l'identifier à celle de type abstrait de structures de données, ou à celle de classe d'arbres syntaxiques ou encore à celle d'«algèbre abstraite». Pour cela, on part de la notion de *structure inductive* (ou «initiale») librement engendrée par une *signature* (ou grammaire abstraite), c'est-à-dire constituée des termes que l'on peut écrire à partir des constructeurs donnés par la signature. À chaque telle structure correspond une théorie logique où le fait qu'elle soit librement engendrée correspond à un schéma de raisonnement par induction (preuve par récurrence). À partir de structures librement engendrées, on peut obtenir de nouvelles structures, plus «contraintes», de deux façons: d'une part en rajoutant des conditions de bonne formation sur les termes (sortes, typage, bon usage des variables, etc.), ce qui équivaut à ne considérer qu'une partie des termes de la structure; et d'autre part en rajoutant des conditions d'égalité sur les termes (par exemple, commutativité ou associativité de certaines opérations), ce qui équivaut à quotienter la structure par une relation d'équivalence. À des fins de pertinence, on s'arrange pour que les constructeurs et les conditions définissant un langage source répondent à quelque critère assez fort d'*effectivité*: calculabilité, récursivité primitive, calculabilité en temps polynomial, etc., de façon que la synthèse, l'analyse, la comparaison ou la copie de termes soient des opérations effectivement simples à réaliser.

Puis, on définit la notion de *correspondance* entre langages comme relation binaire entre les éléments d'un langage et ceux de l'autre. On introduit alors la notion de *sémantique*, comme correspondance entre un langage «concret» et un autre langage «abstrait», qui met en correspondance chaque «signifiant», terme du langage concret, avec ses éventuelles «significations», termes du langage abstrait. On s'intéressera particulièrement aux *sémantiques observationnelles* qui donnent comme signification ab-

straite aux termes concrets l'ensemble des réponses valides à un ensemble d'observations «pertinentes» sur ces termes. On introduira aussi des critères d'effectivité sur les correspondances entre langages et les relations sémantiques, par exemple, dans le cas le plus général de sémantiques observationnelles, la semi-calculabilité de chaque observation.

Un *langage de programmation* est alors la donnée d'un langage source et d'une sémantique permettant de relier ce langage source à un ensemble de comportements observables.

En décrivant ces correspondances entre langages elles-mêmes à l'aide de langages de programmation, on obtient la notion de *métalangage*, dont les termes sont des *métaprogrammes*. On pourra dire de certains métaprogrammes qu'ils traduisent fidèlement un langage dans un autre s'ils préservent la sémantique des termes (si certain diagramme commute). Il est ainsi possible d'exprimer divers critères de correction pour les métaprogrammes, à commencer par interpréteurs et compilateurs. Avec des critères d'effectivité adaptés, on peut donc utiliser une telle infrastructure pour exprimer non seulement des calculs, mais aussi des raisonnements logiques sur les programmes.

Finalement, quand une infrastructure de métaprogrammation est capable de se représenter elle-même, elle est dite réflexive. Une telle infrastructure permet de ne pas avoir à introduire un nouveau métalangage (et tous les axiomes associés) à chaque fois que l'on veut approfondir le comportement du système dans la direction «méta» ; elle brouille aussi la distinction entre programmes et données, et conduit à une vision plus intégrée du développement logiciel⁴.

2.3 programmation multi-aspects

Si nous examinons les applications existantes de la métaprogrammation, nous nous rendons compte qu'elle sert à gérer automatiquement la transition entre plusieurs aspects de mêmes objets informatiques : ainsi, par exemple, quand on compile un programme, on s'intéresse au «même» programme, sous différentes formes (code source ou objet), chacune adaptée à l'effectuation d'opérations différentes (modification par l'homme, ou exécution par une machine).

Conceptuellement, on considère un «même» programme, un «même» objet, une «même» idée, indépendamment des diverses représentations par le truchement desquelles on les communique ou les manipule ; une forme aussi bien qu'une autre peut servir à en décrire l'ensemble des propriétés «intéressantes». Cependant, quelle que soit la représentation choisie, il faut bien en choisir une, et compte tenu des critères d'effectivité imposés par les contraintes physiques et économiques, il vaut mieux en choisir une «adaptée» à réaliser aussi efficacement que possible les manipulations que l'on envisage sur l'objet auquel on s'intéresse.

Or, il est inévitable que les propriétés «intéressantes» d'un objet varient dans l'espace et dans le temps, selon les personnes qui considèrent l'objet, et les composants machines qui le manipulent, selon le domaine d'activité, les attentes, et les compétences des uns, selon le but d'utilisation et les contraintes techniques des autres. Si l'on veut présenter à chaque programme de chaque utilisateur et à chaque moment une représentation des objets considérés la plus adaptée possible à ses intérêts momentanés, il est besoin de métaprogrammes pour assurer la cohérence dans le temps et dans l'espace entre les divers aspects de ces objets.

Ainsi, un avion sera pour un ingénieur s'occupant du fuselage, un ensemble de courbes et d'équations dont il faut optimiser certains paramètres, pour le fabricant de pièces, ce sera un cahier des charges, pour

⁴ La théorie esquissée permet de modéliser formellement la métaprogrammation et la sémantique des métaprogrammes ; cependant, elle ne suffit pas à capturer l'ensemble du phénomène, car le choix de considérer certaines structures plutôt que d'autres comme des «programmes» plutôt que de simples «données» relève dans notre présentation d'une grande part d'arbitraire. En fait, c'est là entièrement une question de point de vue, et seul un observateur extérieur à la modélisation peut disposer d'un critère propre à effectuer «objectivement» cette distinction. Nous verrons plus bas (section 3.2) comment la distinction peut se faire dynamiquement comme sous-produit du processus de développement.

le constructeur, ce sera un processus d'assemblage, pour le responsable de l'entretien, un ensemble de tâches à effectuer, pour le réparateur, un ensemble de problèmes à régler, pour le gestionnaire matériel, un ensemble de pièces détachées, pour le pilote, un appareil à mener à bon port, pour le passager, un inconfort à minimiser pour parvenir à destination, pour le contrôleur aérien, un point à router parmi d'autres, pour l'agent commercial, un ensemble de sièges à remplir, pour le stratège commercial, l'élément d'une flotte à déployer, pour le responsable du personnel, des humains à gérer, pour l'agent comptable, un historique de transactions, pour l'assureur, un risque à évaluer, etc., etc. L'informatisation de tout ou partie du suivi de la vie d'un avion implique de présenter à chacun des acteurs en présence un point de vue de l'avion adapté à ses besoins, et qui contient des paramètres dont la plupart ne sont utiles qu'à lui, mais dont certains concernent aussi d'autres acteurs, avec qui il est essentiel de se mettre d'accord, de synchroniser les données. La métaprogrammation permet d'apporter cette synchronisation, cette cohérence entre les points de vue des multiples acteurs [Kiczales *et al.*, 1997].

Même pour des projets informatiques plus simples que la gestion de A à Z d'une flotte aérienne, il apparaît nécessairement des divergences de points de vue, dès lors que plusieurs acteurs (programmeurs, utilisateurs) sont en présence, et/ou que ceux-ci évoluent, et changent de centres d'intérêt au cours du temps (la même personne pouvant tenir successivement plusieurs rôles). Dès qu'un problème est suffisamment complexe, nul n'a de toute façon les ressources nécessaires pour en embrasser d'un coup l'ensemble de tous les aspects, et il est nécessaire de traiter ceux-ci séparément. Dès lors, la métaprogrammation est utile pour assurer la cohérence entre ces aspects, qui devront sans elle être gérés manuellement ; elle permet de s'occuper une fois pour toutes d'un aspect inintéressant, pour pouvoir l'oublier après.

3 Expressivité de la métaprogrammation

3.1 expressivité et calculabilité

Nous avons présenté la métaprogrammation comme une technique fort utile (et largement utilisée, sans qu'il y en ait conscience) pour résoudre certains problèmes. La question se pose alors de savoir en quoi cette technique est originale, ou ne serait qu'une combinaison (qu'il serait alors intéressant de détailler) de techniques existantes et déjà bien connues, en quoi elle apporte ou non des solutions nouvelles à des problèmes connus ou non encore résolus auparavant, si en fin de compte, c'est une technique essentielle et indispensable, ou si elle est au contraire accessoire.

Cette question est en fait celle de l'expressivité des langages de programmation, en tant qu'algèbres permettant de combiner de multiples techniques : qu'est-ce qui rend un langage plus expressif qu'un autre ? en quoi aide-t-il plus ou moins bien le programmeur à résoudre les problèmes qui sont les siens ? et d'abord, quels sont ces problèmes, et comment les caractériser et les comparer, eux et leurs solutions ?

Le résultat fondamental concernant l'expressivité des systèmes de calcul est celui de Turing [Turing, 1936], qui montre l'existence d'une classe de fonctions dites calculables, capables d'effectuer tout calcul mécaniquement concevable. Étant donné un ensemble d'entrées (questions) possibles et un ensemble de sorties (réponses) possibles, tous deux numérisables (encodables avec, par exemple, les entiers naturels), tout langage de programmation mécaniquement réalisable peut donc exprimer au plus autant de fonctions de l'ensemble des entrées dans l'ensemble des sorties qu'une machine de Turing. Un langage de programmation qui peut exprimer toutes les fonctions calculables d'un ensemble dans l'autre est appelé universel, ou Turing-équivalent (en supposant ces deux ensembles infinis). Les langages universels sont tous aussi puissants les uns que les autres, du point de vue des fonctions qu'ils peuvent exprimer.

Cependant, il est évident que tous les langages Turing-équivalent ne se valent pas du point de vue du programmeur : tous ceux qui en ont fait les expériences respectives conviennent qu'il est plus facile de programmer dans un langage de haut niveau (comme LISP) que dans un langage de bas niveau (comme

C), qui est plus facile d'utilisation que le langage d'assemblage, qui vaut mieux que le code binaire, qui lui-même est plus simple que la fabrication transistor par transistor d'un circuit électronique dédié, ou la spécification d'une machine de Turing. En effet, le résultat de Turing ne constitue que le tout début d'une théorie de l'expressivité des systèmes de calcul, et certainement pas la fin. S'arrêter sur ce simple résultat, et dire «puisque tous les langages, dans la pratique, ne sont pas équivalents comme ils le sont selon Turing, c'est que la théorie n'a rien à dire», ce serait abdiquer la raison et chercher dans un ailleurs vague quelque explication ineffable, ce serait donner la primeur à l'ignorance et la superstition.

Avant d'aller plus loin, notons que la démonstration du résultat de Turing est toute entière fondée sur la métaprogrammation : si les langages universels sont équivalents les uns aux autres, c'est parce qu'il est toujours possible d'écrire dans n'importe lequel de ces langages un interpréteur pour tout autre langage, si bien que tout programme dans l'autre langage peut trouver, modulo l'usage d'un traducteur, un équivalent dans le langage de départ. C'est d'ailleurs pour cela que ces langages sont appelés *universels*. Or, la métaprogrammation est un style de programmation ignoré de toutes les méthodes de développement logiciel appliquées à la plupart des langages⁵, car elle consiste précisément à ne pas utiliser le langage étudié, mais un autre langage, via des métaprogrammes. Cette réticence vis-à-vis de la métaprogrammation peut-elle être formalisée, et que reste-t-il alors de l'équivalence entre langages universels ?

3.2 le processus compte

Il n'existe malheureusement pas encore de théorie pleinement satisfaisante de l'expressivité ; le seul travail récent que nous avons pu trouver sur le sujet [Felleisen, 1991], fort intéressant par ailleurs, limite son ambition à la macro-expressibilité relative d'extensions d'un même langage l'une dans l'autre. Cependant, nous disposons de quelques morceaux épars d'une théorie générale, qui suffisent amplement selon nous d'une part à justifier cette idée somme toute «intuitive» que la métaprogrammation apporte un surcroît d'expressivité, et d'autre part à raffiner ou rejeter d'aucunes affirmations communément entendues sur l'expressivité «excessive» de certains langages.

Le point d'achoppement sur lequel la notion de calculabilité ne suffit pas à exprimer(!) l'expressivité des langages de programmation, c'est que le développement de programmes informatiques ne se résume pas à l'écriture mono-bloc (ou l'échec d'une telle écriture) d'une solution parfaite à un problème statique donné, mais un processus dynamique et évolutif, faisant intervenir une interaction plus ou moins poussée entre l'homme et la machine.

Ainsi, pour tout programme donné à écrire sur une machine donnée, la solution véritablement optimale ne pourra s'exprimer qu'en langage binaire, et contiendra des tas de «trucs» de derrière les fagots, d'encodages basés sur des coïncidences heureuses, de «jeux de mots» sur l'encodage des données, des adresses, et des instructions machines. Mais le fait est que trouver une telle solution demanderait des forces titanesques, et l'apparition de nouvelles architectures matérielles rendront obsolètes toutes ces optimisations jeux-de-mots. Or les forces humaines dépensées au cours du développement sont importantes ; elles sont même essentielles, vu qu'au bout du compte, le but de l'ordinateur, comme de tout outil, est de minimiser la quantité d'efforts humains nécessaires à l'accomplissement de tâches de plus en plus élaborées. Si le coût relatif des ressources humaines et mécaniques était tel, dans les années 50, que la production de code binaire super-optimisé à la main était économiquement viable (voir l'histoire de Mel extraite du Jargon File [Raymond, 1996]) ; une telle production est impossible aujourd'hui.

Le problème n'est donc pas seulement technique, il est aussi économique, moral et politique, en ce qu'il concerne des déplacements d'efforts humains. Dans l'écriture de programmes informatiques, comme partout ailleurs, *le processus compte*. Et c'est bien ce processus qu'essaient d'améliorer sans le formaliser

⁵ Voir plus loin cependant le cas des familles de langages FORTH et LISP, ainsi dans une certaine mesure celui des langages fonctionnels «statiquement typés».

rationnellement les notions courues de réutilisation de code, de modularité, de programmation dynamique ou incrémentale, de méthode de développement, etc. De même, la tendance en informatique à abandonner les langages de trop bas niveau, en faveur de langages de plus haut niveau tient précisément à ce que l'intelligence humaine est une ressource limitée, voire rare, et qu'il faut la réserver pour les tâches les plus importantes, celles où elle est indispensable (sinon à jamais, du moins à cette heure).

Aussi, une modélisation satisfaisante de l'expressivité des langages de programmation, même assez abstraite pour ne pas dépendre excessivement de considérations technologiques éphémères, se doit de prendre en compte l'interaction homme-machine, et d'une façon qui inclue une notion de coût humain (et peut-être aussi des notions d'erreur et de confiance). Nous soupçonnons qu'une telle modélisation est possible en utilisant la théorie des jeux.

3.3 complexité de la programmation incrémentale

À défaut d'une théorisation algébrique cohérente de l'expressivité, nous pouvons en donner une première approximation intuitive suivante, prenant en compte une notion bête et méchante du coût humain : un système de programmation sera plus adapté à un certain usage qu'un autre, s'il nécessite «à la longue» moins d'interaction humaine pour résoudre successivement une série indéfinie de problèmes posés dans le domaine donné. On pourra utiliser les «métriques» approximatives courantes de production (nombre de lignes de code, ou de constructions syntaxiques abstraites, ou de symboles écrits, nombre de mois-hommes bloqués). Sans fournir de caractérisation fine de l'expressivité (qui serait nécessaire à la définition d'un style de programmation conséquent), cette approximation suffit à justifier l'usage de la métaprogrammation.

Dans le processus de développement traditionnel, d'où la métaprogrammation est exclue, la quasi-intégralité du code constituant les logiciels est écrite manuellement par l'humain ; le détail de l'exécution, l'ensemble du comportement, tout doit découler directement de la pensée humaine ; à chaque étape de développement, toute modification est l'œuvre de l'homme. Là où le bât blesse, c'est que tôt ou tard, certaines nouvelles fonctionnalités à ajouter à un projet logiciel nécessitent des changements architecturaux globaux plus ou moins profonds : toutes les modifications, globales, du programme devront alors être effectuées à la main, avec tous les problèmes d'incohérences involontaires introduites par de tels changements, qui induisent de nombreux bogues et coûtent de nombreuses itérations de développement (pour un tel phénomène publiquement documenté, voir les changements occasionnels d'API du noyau Linux ; pour un exemple spectaculaire, voir le bogue explosif de la première fusée d'Ariane V). L'alternative de ces modifications est la réimplémentation complète, qui coûte tout ce que coûte la réécriture de code. Finalement, le coût incrémental de développement traditionnel est proportionnel à la taille de code différent entre deux itérations du processus.

Faisons entrer la métaprogrammation dans le cycle de développement. La métaprogrammation permettant des traitements arbitraires du code par la machine, il devient possible au programmeur faisant face à un changement structurel de son programme de faire effectuer semi-automatiquement toute tâche d'instrumentation ou de transformation de son programme, pour le mettre en conformité avec la nouvelle architecture, pour vérifier le nouveau code vis-à-vis des invariants déclarés, pour assurer la cohérence des différentes parties de son programme. «Semi-automatiquement» signifie ici que le «métaprogramme» le plus simple pour gérer certains cas particuliers consiste parfois à les traiter manuellement au cas par cas. Bref, entre chaque itération du processus de développement logiciel, le coût incrémental est proportionnel non pas à la taille de code différent entre les états successifs du projet, mais à la taille du plus petit métaprogramme permettant de transformer l'état précédent du projet en l'état suivant, c'est-à-dire à la complexité de Kolmogorov conditionnelle [Li and Vitanyi, 1998] d'un état par rapport au précédent !

Dans le pire des cas, la métaprogrammation n'aura pas servi, et le coût sera exactement le même (à

une constante additive négligeable près) que celui de la programmation traditionnelle⁶ ; la théorie nous enseigne que, par la définition même du concept d'aléatoire, cela correspond au cas où l'évolution du projet est complètement aléatoire par rapport à la base de code existante. Dans le cas le meilleur, un gain arbitraire a pu être fait. En général, la métaprogrammation permet un coût optimum du point de vue de la complexité ; ses gains par rapport à l'approche traditionnelle sont proportionnels à la taille du projet, dans le cas inévitable où des changements structurels à effets globaux sont nécessaires.

Ainsi, la théorie de la complexité de Kolmogorov nous permet donc de voir que la métaprogrammation gagne toujours, et souvent de beaucoup, sur une programmation non méta, en optimisant le partage d'information entre itérations du processus de développement. On voit bien d'ailleurs combien les doses homéopathiques de métaprogrammation utilisée dans les projets logiciels traditionnels sont un facteur déterminant pour le déroulement de ces projets : choix, souvent statique, d'un environnement de développement ; filtrage des programmes au moyen d'outils d'analyse statique, vérificateurs d'invariants plus ou moins élaborés et autres `lint` ; usage d'éditeurs gérant l'indentation et la structure, qu'ils peuvent mettre en valeur avec fontes et couleurs ; utilisation occasionnelle d'outils de recherche et remplacement sur le source du programme ; dans le cas de bidouilleurs⁷ plus audacieux, «macros» ou «scripts» écrits en *elisp*, *perl* ou autre, pour éditer et manipuler les sources.

Cependant, et ceci va se révéler très important pour la suite de notre exposé, la métaprogrammation gagne si et seulement si d'une part un effort infrastructurel est effectué pour permettre une manipulation aisée des programmes, et si d'autre part il y a une assez grande redondance entre développements successifs pour permettre un partage non trivial d'information au cours du temps ; le bénéfice permis par la métaprogrammation est donc asymptotique : il est faible au départ, mais se bonifie grandement avec la taille des programmes et le nombre d'itérations de développement, c'est-à-dire avec l'*extension* spatio-temporelle des projets de développement logiciel.

4 Disponibilité des sources

4.1 métaprogrammation contre propriété intellectuelle

Admettons l'intérêt des techniques liées à la métaprogrammation. Pourquoi ne sont-elles pas plus communément connues et consciemment utilisées ? Y aurait-il donc des freins qui ont empêché jusqu'ici leur plus large diffusion ?

Il nous semble quant à nous évident que les barrières à la diffusion des sources sont autant de freins au développement de la métaprogrammation. En effet, la condition même d'utilisation d'un métaprogramme lisant un programme en entrée est la disponibilité d'un programme à lire ; la condition d'utilité d'un métaprogramme écrivant un programme en sortie est que ledit programme puisse être diffusé et utilisé ; et ces conditions se combinent si le métaprogramme lit et écrit à la fois des programmes, et plus encore s'il dépend de l'accumulation sur le long terme de connaissances sur les programmes ! Toute limitation sur les manipulations de programmes limite d'autant la faisabilité ou l'intérêt des métaprogrammes, et décourage les métaprogrammeurs potentiels.

Or quelles sont les implications du régime actuel de «propriété intellectuelle» sur les programmes ?

D'abord il y a le droit dit «d'auteur», mais en fait, d'éditeur, puisque tout auteur abandonne tous ses droits à l'entreprise qui l'embauche ou qui l'édite (à moins de se faire lui-même éditeur ; mais il gagnera son argent non en tant qu'auteur mais en tant qu'éditeur, ce qui n'enlève rien au problème). Pour

⁶ De toute façon, la métaprogrammation étant une extension de la programmation traditionnelle, et offrant une panoplie d'outils supplémentaires sans retirer aucune option existante, elle ne peut faire qu'au moins aussi bien que la programmation traditionnelle, en terme de minimum de travail nécessaire pour accomplir une tâche.

⁷ traduction correcte à notre avis du terme «hacker».

respecter ce «droit», un métaprogramme doit refuser de faire certaines inférences, certaines manipulations, qui violeraient (selon les pays) la licence sous laquelle le programme d'entrée est disponible. Quelles manipulations exactement sont interdites, la chose n'est pas claire.

Ainsi, certaines licences sont par utilisateur, ce qui interdit à un métaprogramme toute inférence qui soit utile à deux utilisateurs (ou à un utilisateur non autorisé) pour des notions d'utile et d'utilisateur restant à préciser (quid si deux personnes collaborent sur un même document ? ou si l'une porte assistance à l'autre ou lui sert de secrétaire ? quid si le résultat final du calcul est distribué à des millions d'individus ?). Si elles sont par machine, ou par utilisateur *à la fois*, le problème se complique, parce que les notions invoquées par ces licences deviennent inopérantes en présence de machines à temps partagé, de machines à multiprocesseurs, de machines en grappes, voire, pire, de machines plus rapides : aucun paramètre logiciel ne peut correspondre à ces notions, et encore moins avec des émulateurs logiciels d'architecture (émulateurs 680x0 ou x86), et pis encore si ces émulateurs font appel à des techniques d'analyse et de traduction de code (elles aussi prohibées par de nombreuses licences) qui dénaturent toute correspondance précise entre d'une part la machine «abstraite» pour laquelle le code est écrit et la licence prévue, et d'autre part la machine «concrète» sur laquelle tourne le programme. La notion d'utilisateur devient floue aussi face à des métaprogrammes qui utilisent automatiquement un programme donné, et le deviendra sans doute de plus en plus avec l'émergence souhaitée d'«intelligences artificielles». Tous ces problèmes peuvent être évités au prix énorme de n'utiliser en entrée des métaprogrammes que des logiciels et des informations libres de droits, ce qui, pour rester légaliste, requiert de faire signer un papier (ou cocher une case dans un menu ?) à toute personne introduisant au clavier quelque information originale.

Un problème plus épineux, que cette mesure ne résout pas, est la diffusion des résultats de métaprogrammes. Car, même avec des licences de logiciel libre, qui peuvent être incompatibles l'une avec l'autre, et plus encore avec des licences exclusives, certaines opérations sont permises, mais sous couvert de non-diffusion du résultat («usage privé»). Mais qu'est-ce exactement qui ne doit pas être transmis, quand un métaprogramme a inclus dans sa base de donnée persistante la synthèse des analyses de nombreux programmes ? Qu'est-ce que redistribuer ? Cela s'applique-t-il entre personnes d'une même entreprise ou entité morale ? Quid si cette entité est une association admettant toute personne comme membre ? Cela s'applique-t-il aux machines ? aux composants internes d'une même machine ? Et si cette même machine est une grappe en regroupant plusieurs ? Et si cette grappe recouvre le monde entier ?

Le fin du fin est l'existence de brevets. Ceux-ci affectent les métaprogrammes universellement, indépendamment du matériau de base utilisé et de leurs licences. Ils n'introduisent aucun règle structurelle interdisant une inférence logique dans un métaprogramme ; mais en revanche, si à un moment donné, le métaprogramme doit entraîner l'utilisation de certains algorithmes dans un but donné, alors l'inférence ayant entraîné ce résultat doit être rétroactivement invalidée, à moins que ne soit obtenue une licence auprès du détenteur de brevet. Mais comme les notions d'algorithme et de but sont on ne peut plus floues, il faudra un méta-méta-programme on ne peut plus élaboré, pour surveiller que le métaprogramme n'enfreint jamais aucun des millions de brevets déposés.

Si toutes les règles ci-dessus portant sur l'usage de métaprogrammes vous paraissent ridiculement absurdes, n'oubliez pas qu'après tout, les «intelligences artificielles», si elles diffèrent grandement dans leur qualité actuelle des «intelligences naturelles», sont basées sur les mêmes principes, et sur les mêmes contraintes. Ainsi, sachez que le métaprogramme le plus répandu actuellement, à savoir l'esprit humain, est le premier astreint à ces absurdes contraintes, par les mêmes absurdes lois !

On conçoit, dans ces conditions, que seuls les métaprogrammes les plus simplissimes aient pu être assez utilisables et utilisés pour être réellement développés. Point d'intelligence artificielle digne de ce nom en vue, ou de base de données universelle, tout effort coopératif d'automatisation étant constamment freiné

par les barrières de «propriété intellectuelle»⁸.

L'écriture et l'utilisation de métaprogrammes conformes avec la législation en vigueur est infernale. Il n'y a aucune manière logiquement cohérente de définir des règles pour ou contre l'utilisation des métaprogrammes, même les plus simples, en présence de «propriété intellectuelle». Aussi la seule loi est-elle celle du plus fort, qui est celui dont les millions peuvent acheter les meilleurs avocats pour faire valoir ses «droits». Nous pensons qu'il n'est pas possible de définir une quelconque notion de «propriété intellectuelle» qui soit techniquement applicable dans un cadre légal, et ne soit pas profondément injuste et nuisible économiquement ; nous avons dans un autre document [Rideau, 1997] donné une argumentation complète en faveur de ce point de vue. Toutefois, si une telle notion existait, c'est aux partisans de cette «propriété intellectuelle» que revient la charge d'en exposer une définition, et de prouver sans équivoque possible son caractère bénéfique.

En fait, tant que la seule opération menant à la production de code est l'ajout manuel, l'écriture à partir du néant par un esprit humain supposé inspiré directement par les Muses, alors il est possible d'attribuer à chaque mot, à chaque symbole composant un programme une «origine», un «auteur». Mais dès que l'on autorise la métaprogrammation, c'est-à-dire des opérations arbitraires agissant sur le code et produisant du code, dès lors que l'on considère l'environnement dans lequel baigne l'auteur (homme ou machine) d'un programme, alors celui-ci n'est plus l'inventeur du programme, mais seulement le dernier maillon d'un processus holistique de transformation, où il n'est pas possible d'attribuer une origine univoque à quelque élément que ce soit.

4.2 cloisonnement et déresponsabilisation

Par delà les considérations technico-légales précédentes, le système de développement logiciel exclusif («propriétaire») est tout entier, corps et âme, opposé à la métaprogrammation.

Dans le processus de développement fermé traditionnel, les logiciels sont écrits de manière cloisonnée, à l'intérieur des barrières constituées par les licences exclusives. Chaque développeur est seulement de passage dans le développement d'un logiciel sur la destinée duquel il n'a aucun contrôle, qui ne le concerne

⁸ Encore une fois, les arguments en faveur de la liberté d'utilisation et de distribution des informations sont de même nature que ceux en faveur du libre échange [Bastiat, 1850]. Les barrières de «propriété intellectuelle» valent bien les barrières douanières aux frontières des pays, à l'entrée ou à la sortie des villes, au passage des routes et des ponts ; les unes comme les autres s'opposent au libre échange des biens et des services ; les unes comme les autres sont aussi sûrement artificielles qu'elles nécessitent pour être respectées l'emploi de la force armée à l'encontre de citoyens se livrant de manière consentante à des échanges ne lésant les ressources de nul tiers.

La création, la recherche, la distribution, l'enseignement, la correction, la garantie, le maintien à jour d'informations sont des services, que les hommes sont naturellement portés à s'échanger contre d'autres services. Les barrières de «propriété intellectuelle» sont autant d'obstacles à l'échange de ces services, et établissent le monopole d'un éditeur sur l'ensemble des services associés à chaque information rendue «propriétaire». Le marché des services informationnels en devient encore plus fractionné que le marché des biens au Moyen Âge, car les barrières sont maintenant autour de chaque personne, de chaque compagnie, de chaque institution, de chaque éditeur, plutôt que seulement autour des villes, des routes, des ponts et des pays. Le consommateur et l'entrepreneur sont les victimes permanentes de cet état de fait auquel la loi, malheureusement, concourt.

Ce protectionnisme en matière de services informationnels, ô combien nuisible à l'industrie, n'est aucunement justifiable par un droit naturel à la propriété intellectuelle. En effet, si les biens, en tant que ressources physiques limitées en extension, sont naturellement sujets à être contrôlés et possédés, l'information est intrinsèquement et indéfiniment duplicable et partageable, et ne se prête pas au moindre contrôle, à la moindre possession. C'est du reste bien la raison pour laquelle d'aucuns réclament *de la loi* l'établissement d'une «propriété intellectuelle» artificielle, alors que la propriété matérielle précède de loin la loi, qui ne fait que la confirmer. Nous voyons bien que ceux-là ne font que demander, comme tout protectionniste, la spoliation légale et doublement coûteuse de l'ensemble des citoyens-consommateurs à leur profit personnel ou corporatif de producteurs.

La libre circulation de l'information, enfin, est nécessaire à la définition de conditions équitables dans les échanges. Elle est donc nécessaire au bon équilibre des marchés économiques, et toute atteinte à cette libre circulation biaise les marchés, crée des bulles financières, et engendre la ruine.

pas, qui est souvent destiné à finir dans un grenier (et lui à la porte) à moyen terme par décision marketing, au grand dam des utilisateurs. Ces derniers, n'ayant aucun accès, direct ou indirect, aux sources, sont complètement sans défense. Chaque groupe de développeurs n'a accès qu'à une petite partie du tout qui constitue l'ensemble du système logiciel tel qu'il fonctionnera, et, comme cela nous intéresse surtout, tel qu'il tourne lors du développement. Ainsi, les développeurs n'ont pas accès à l'infrastructure au-dessus de laquelle ils écrivent leurs programmes ; si celle-ci se révèle insuffisante, ils doivent accepter ses limitations et contourner celles qu'ils peuvent contourner, ou la réécrire entièrement à partir de rien, à moins de trouver quelque moyen pour la faire modifier par ses auteurs.

L'écriture ou la réécriture d'outils de développement coûte fort cher en ressources matérielles et intellectuelles de développement ; au cas où l'infrastructure réécrite demeure elle-même exclusive, conformément au modèle de développement ambiant, elle mène à renforcer le cloisonnement des développeurs, en introduisant des outils de développement incompatibles auxquels il faut former tout nouveau collaborateur. Ces outils, de plus, sont souvent fort imparfaitement conçus, par des personnes dont ce n'est pas la vocation première, qui n'ont pas la formation ou la culture adéquate, et ignorent souvent l'état de l'art de la recherche scientifique sur le sujet, état de l'art qu'ils n'ont de toute façon pas les ressources pour suivre et encore moins mettre en œuvre. Dans le monde du logiciel exclusif, l'invention d'un nouveau langage ou outil est donc une gageure risquée et excessivement coûteuse. Sauf s'il a les moyens d'être maître d'un large marché captif, de façon à pouvoir imposer la direction suivie par l'évolution de ses produits «méta», l'inventeur d'un outil à vocation universelle, ou un tant soit peu générale, aura donc tout intérêt, *en tant qu'utilisateur* de ce métaprogramme, à briser la logique de cloisonnement et à le diffuser librement⁹.

Une autre possibilité qui s'offre aux utilisateurs insatisfaits d'une infrastructure exclusive consiste à infléchir la position des fournisseurs de l'infrastructure ; cependant, il n'y a pas pour cela de solution fiable, durable ou profonde, à moins d'acheter ces fournisseurs eux-mêmes, au prix de millions de dollars et de réorganisations administratives. Enfin, il leur reste la possibilité de participer à des comités de standardisation, qui permettent un certain retour d'information de la part des utilisateurs, nonobstant les lenteurs administratives et les arguties politiques. Mais dans le cadre de la normalisation de logiciels dont les fournisseurs majeurs suivent le modèle exclusif, les souhaits des utilisateurs prennent d'autant moins de place que le poids capitalistique et technologique, donc décisionnel, est concentré dans les mains des fournisseurs. Nulle décision ne saurait être d'ailleurs prise sans que les fournisseurs n'en aient déjà les éléments d'implémentation et donc aient déjà pris les décisions majeures sur l'évolution de leurs produits. Certes, les fournisseurs font évoluer leur produit pour satisfaire leur clientèle, mais entre les besoins réels de l'utilisateur et la décision finale se trouvent assez d'intermédiaires et d'interprètes infidèles, pour que ces besoins aient une importance marginale, en fin de compte.

Le bénéfice de la métaprogrammation est, nous l'avons vu, dans la gestion automatique des redondances du code dans l'espace et le temps. Le cloisonnement, en restreignant le partage spatio-temporel des programmes sources, empêche les programmeurs de confronter le code résultant de leurs expériences respectives, réduit les occasions de détecter des redondances, de simplifier le code, d'automatiser la propagation d'informations ; il bride ou anéantit le bénéfice de la métaprogrammation, et incite à davantage de programmation de bas niveau, à l'écriture de programmes jetables, sans passé ni futur.

Quelles sont les conséquences de ce cloisonnement sur le code lui-même ? Les développeurs des différents composants sont déresponsabilisés les uns vis-à-vis des autres, et tout particulièrement des différents niveaux «méta». Ils ne poussent jamais une approche métaprogrammative de développement d'outils assez puissants et génériques, car, isolés, ils auraient à en payer l'intégralité du coût, pour une partie minime des avantages. Ils interagissent donc par des interfaces (API) grossières, superposées en «couches»

⁹ Il y a ainsi au moins un projet commercial, qui, étant résolument orienté dans une démarche de métaprogrammation, a suivi cette voie : le projet de développement de plates-formes mobiles réparties de chez Ericsson, pour lequel a été spécifiquement développé le langage et l'infrastructure Erlang/OTP, aujourd'hui publiés comme logiciel libre <http://www.erlang.org/>.

hermétiques non moins grossières. Ces API, pour être commercialisées, devant être fondées sur des standards indépendants de leurs outils de développement exclusifs, ils les construisent, par un nivellement vers le bas, sur des «plus petit commun dénominateurs» extrêmement précaires, dénuées de tout pouvoir d'abstraction (bibliothèques interfacées avec C ou C++ ; flots d'octets ; CORBA) ; la majeure partie de leur sémantique ne peut être exprimée dans le langage informatisé de l'interface, et doit être expliquée informellement dans la documentation ; toute vérification sur le bon usage de ces API doit donc se faire à la main, ce qui rend erreurs et incohérences non-détectées très fréquentes. Les développeurs n'ayant pas de contrôle sur les langages qu'ils utilisent et leur implémentation, ces langages évoluent peu ou mal, guidés par de vagues «benchmarks», qui par effet thermostat, encouragent les programmeurs à s'adapter encore et plus aux styles de programmation «reconnus», indépendamment de leur qualité technique. Les fournisseurs d'outils exclusifs, une fois qu'ils ont atteint le «standard» du marché, étendent leurs outils pour couvrir une très large gamme de besoins marginaux, mais assez spectaculaires pour impressionner les gestionnaires non-techniciens, mais laissent le plus souvent (sauf par hasard) de côté l'essentiel, qui répondrait aux besoins profonds des développeurs.

Le fruit amer de l'adoption de ce paradigme de cloisonnement par l'industrie est donc une culture de la déresponsabilisation des développeurs. Les programmeurs, sans moyen d'intervention, apprennent à accepter sans esprit critique, sans la moindre notion même basique d'appréciation, les outils de développement et les langages de programmation, qui leur sont en fin de compte imposés par une direction qui n'a aucune compétence pour juger de leur valeur. Toute «décision du marché» en la matière de langage de programmation se confirme alors d'elle-même, indépendamment de tout critère technique direct, influencé principalement par l'inertie de l'existant, et le matraquage intellectuel issu des services marketing des fournisseurs. Ainsi ont pu s'imposer des langages comme C++ ou Java, résolument inférieurs à l'état de l'art en matière de langages de programmation.

Ainsi aussi s'imposent, par effet auto-stabilisant, des monopoles sur chaque segment du marché, le monopole le plus agressif mangeant les autres, et empêchant tout progrès de s'effectuer sans lui. Certes, à l'intérieur de ces monopoles, le partage des ressources, s'il a lieu, peut engendrer de l'intérêt pour la métaprogrammation ; mais l'absence de concurrence ne pousse pas à l'utiliser, pas plus que n'y incite l'état de déresponsabilisation générale des informaticiens recrutés depuis l'extérieur, ou la nécessité pour toute communication avec l'extérieur du monopole de suivre des formats fermés et opaques, prévus pour empêcher la métaprogrammation. Même si chaque monopole peut profiter dans une certaine mesure de techniques métaprogrammatives de manière interne, le développement cloisonné a tout de même pour résultat une désaffection vis-à-vis de la métaprogrammation, et des dégâts matériels et moraux considérables sur l'industrie en général.

4.3 processus de développement ouvert

Si le paradigme de développement cloisonné de logiciel exclusif a pour conséquence la désaffection vis-à-vis de la métaprogrammation et la corruption même des outils de développement *e contrario*, l'émergence de l'informatique libre, où les logiciels sont développés de manière ouverte, avec disponibilité complète des sources, entraînera la responsabilisation des programmeurs et apportera la qualité des outils de développement. Ces effets à eux seuls justifient le succès grandissant du modèle de développement libre, tel qu'actuellement observé.

Dans un processus de développement ouvert [Raymond, 1997] nulle barrière, nulle limitation arbitraire, nul diktat irresponsable de la direction ; la qualité compte avant tout. Car les efforts de la communauté des développeurs se porteront naturellement¹⁰ sur les bases de code les plus adaptées, qu'il sera plus aisé

¹⁰ «Naturellement», c'est-à-dire si elle est suffisamment bien informée, et ne subit pas trop les contrecoups du matraquage intellectuel et du déplacement de capitaux dus au modèle de développement ambiant.

d'étendre et de maintenir. Les développeurs sont rendus responsables et des logiciels qu'ils produisent, et de ceux qu'ils utilisent.

Ainsi, et bien que peu en aient pleine conscience, ce phénomène intervient aussi au niveau méta, par le choix et le développement d'outils de développements libres. Ce n'est pas par hasard que le monde du libre logiciel a engendré de nombreux nouveaux outils de développement que leurs utilisateurs n'hésitent pas à étendre, par delà tout standard existant mais de manière ouverte, créant par là même de nouveaux standards : ainsi les langages et/ou implémentations Emacs LISP, GNU C, Perl, Python, GUILE (et de nombreuses autres implémentations de Scheme), Tcl, bash, zsh, pour ne citer que les plus populaires, de même que des langages sortis tout droit du monde académique comme Objective CAML, Haskell, Mercury, etc. ; mais aussi, il ne faut pas les oublier, les outils tels que (X)Emacs, GNU make, CVS, diff, patch, rsync, et en fait, tout ce qui constitue les systèmes Unix libres (GNU/Linux, BSD), et avec eux l'infrastructure de l'Internet.

Les utilisateurs ont une part active, voire interactive, dans l'élaboration de ces outils et de leurs implémentations : ils suggèrent des améliorations ou extensions, dénoncent les erreurs, et le cas échéant, se collent eux-mêmes à l'implémentation de ces améliorations ou extensions, et corrigent eux-mêmes ces bogues. C'est là le niveau zéro de la métaprogrammation, complètement absent pourtant du modèle de développement logiciel exclusif. Reste à savoir si le libre logiciel aura tendance à en rester là concernant la métaprogrammation, ou s'il va la porter à des niveaux jamais encore atteints aujourd'hui.

Or, puisque l'étude du processus de développement milite fortement en faveur de la métaprogrammation pour des raisons de complexité, et que métaprogrammation et libre disponibilité des sources semblent être intimement liées, voyons si un argument simple sur ce processus de développement ne justifie pas à la fois métaprogrammation et développement ouvert. Ainsi, de même que la justification de la métaprogrammation se trouvait dans l'optimisation du partage d'information dans le processus de développement, en prenant en compte son extension dans le temps, la justification du développement ouvert se trouve dans l'optimisation du partage d'information dans le processus de développement, en prenant en compte son extension dans l'espace. À partir du moment où on considère ce processus dans son extension spatio-temporelle, et sachant que temps et espace sont ultimement convertibles l'un en l'autre, il devient clair que les deux se rejoignent : qu'un programmeur communique avec un lui-même futur, ou avec un autre programmeur, reste le problème du contrôle, de la décision dans un monde où interagissent de multiples acteurs. On comprend qu'à ce titre, l'avènement de l'Internet, qui multiplie la vitesse et la bande passante avec lesquelles les développeurs potentiels s'échangent des messages amplifie l'utilité voire la nécessité de méthodes pour traiter efficacement l'information partagée, à savoir la métaprogrammation et le développement ouvert¹¹.

Ainsi, non seulement métaprogrammation et libre disponibilité des sources sont liés, au sens que l'un permet de tirer parti de l'autre d'une façon qui n'est pas autrement possible, mais encore ce lien est profond et structurel : métaprogrammation et développement ouvert sont en fait deux aspects d'un même phénomène qui les unifie : la programmation réflexive (appellation que nous allons justifier), qui optimise le partage d'information dans le processus de développement, aussi bien dans le temps que dans l'espace.

¹¹ Ainsi, l'Internet est à la fois le vecteur et l'aboutissement d'une coopération mondiale de bidouilleurs autour des sources librement disponibles.

5 Programmation réflexive

5.1 systèmes réflexifs

Examinons les propriétés d'une plate-forme réflexive de développement, qui intégrerait aussi bien métaprogrammation que libre disponibilité des sources. Une telle plate-forme contiendrait tous les outils nécessaires à la programmation et la métaprogrammation, et les sources desdits outils seraient non seulement librement disponibles dans leur intégralité, mais aussi conçus pour être à leur tour traités semi-automatiquement, par le système, sous la direction des développeurs. Le mot réflexif est ainsi justifié, parce que les outils du système se manipulent en quelque sorte «eux-mêmes»¹².

Les systèmes Unix libres (GNU/Linux, *BSD, etc.) et certains autres (comme Native Oberon) sont d'une certaine façon autant de systèmes réflexifs complets : ils sont libres, et contiennent avec leurs sources tous les outils logiciels nécessaires à leur auto-développement, de l'interface utilisateur au compilateur et aux gestionnaires de périphériques. Mais fort longue est leur «boucle réflexive» qui identifie programmeur et métaprogrammeur. Si on considère comme «acquis» le système d'exploitation et ses outils de développement de base (y compris un compilateur C), alors, les nombreuses implémentations de langages de programmation qui sont écrites dans le langage implémenté lui-même, sont tout aussi «réflexives», et avec une boucle réflexive courte entre implémentation et utilisation ; toutefois, ces implémentations ne constituent pas des systèmes complets (et cherchent rarement à en constituer), puisqu'elles dépendent pour leur développement de nombreux services extérieurs.

Maintenant, ces systèmes réflexifs existants, complets ou pas, ne tirent qu'à peine parti de leur aspect réflexif, qui reste essentiellement implémentatoire : les sources ne sont pas conçus pour être manipulés par autre chose que par les compilateurs ou interpréteurs des langages utilisés, ni pour être produits par autre chose qu'un éditeur de texte. Les utilisations de la réflexion à fins d'analyse ou de synthèse sont à ce jour très limitées et découplées les unes des autres, et ne font appel à aucun processus persistant ou accumulatif. De plus, les langages utilisés ne se prêtent pas aisément à la métaprogrammation ; ils sont conçus pour fonctionner dans les systèmes fermés existants, et imprégnés d'une culture de développement cloisonné, excluant toute métaprogrammation ; parfois, ils datent de l'époque héroïque où la métaprogrammation était extrêmement limitée par la faible capacité calculatoire et mémorielle des machines.

Il reste donc encore à développer une plate-forme de développement authentiquement réflexive, où la métaprogrammation sous toutes ses formes sera intégrée à un processus de développement ouvert. Il existe, à notre connaissance, un seul projet résolument orienté dans cette direction¹³ mais il est encore balbutiant, car il lui manque sans doute le cœur du système : un langage de programmation adapté à un style de développement réflexif.

5.2 style réflexif

À quoi ressembleront les systèmes informatiques, réflexifs, de demain ? Quel style réflexif de programmation conviendra au développement logiciel sur de telles plates-formes ? Comment l'art de programmer sera-t-il affecté par l'impact grandissant de la métaprogrammation, utilisée à grande échelle sur des sources libérés ?

La Turing-équivalence assure que quel que soit le langage choisi au départ, la métaprogrammation permet le développement renouvelé d'outils aussi adaptés que possibles aux domaines explorés par les

¹² Paradoxe oblige, cette affirmation demande certaines précautions pour être formalisée rigoureusement, qui sortent de l'objet de cet essai ; nous préparons un rapport technique consacré à cet exercice.

¹³ Le projet *Tunes*, <http://www.tunes.org/>

programmeurs : c'est là le problème de l'«amorçage» bien connu de tous les implémenteurs de compilateurs écrits dans leur propre langage cible. Bref, elle assure qu'un processus de développement libre mènera fatalement à la métaprogrammation, et, sur le long terme, à un système réflexif. Mais elle ne donne aucune indication sur ce à quoi doit ressembler un système réflexif ayant atteint sa maturité.

Une telle indication se trouve sans doute plutôt dans l'expérience de systèmes de développement dynamiques existants, comme les langages des familles FORTH ou LISP (ou leur croisement, le RPL), qui possèdent une boucle d'interaction edit-compile-run-debug pratiquement instantanée, et ont intégré des techniques originales et variées de métaprogrammation dans leur pratique quotidienne : évaluation interne, échappements permettant d'exécuter du code «immédiatement» au cours de la compilation, «macrodéfinitions» arbitrairement puissantes, et autres «protocoles à méta-objets». Ces techniques ont été utilisées, entre autres, pour implémenter de manière interne à ces langages, et avec une sophistication parfois extrême, divers paradigmes de programmation (systèmes à objets, programmation logique, évaluation concurrente, etc.), extensions de langage et autres sous-langages dédiés [Shivers, 1996]. Les systèmes construits autour des ces langages suggèrent qu'un système réflexif serait «intégré», toutes les fonctions étant utilisables à partir du même langage, en minimisant ainsi la redondance des efforts linguistiques entre méta-niveaux, et en justifiant l'appellation de «réflexif» par rapport aux remarques de la section 2.2.

Ces langages sont indubitablement à ce titre les plus expressifs actuellement disponibles, et nous conjecturons qu'utilisés dans cette optique, ils sont capables d'offrir la plus grande productivité. Toutefois, on ne peut que constater qu'ils n'ont pas «percé» outre mesure dans l'industrie, quoiqu'ils aient une longévité remarquable, et aient séduit techniquement ceux qui les ont essayés. Outre les raisons générales déjà évoquées au sujet de l'effet du développement cloisonné sur l'industrie, il nous semble que, paradoxalement, un style réflexif est un handicap plutôt qu'un avantage pour des langages fournis dans le cadre du logiciel exclusif cloisonné : une grande intégration à un logiciel de développement fermé a de trop nombreux coûts, directs et indirects, pour l'utilisateur, sous forme de limitations dans le déploiement d'abord (choix de matériels restreint, licences, royalties), mais surtout dans la subordination de forts investissements techniques et humains à une technologie dont il ne maîtrise pas l'évolution, puisque seul le propriétaire de l'outil de développement a la possibilité de le faire évoluer ou même seulement survivre. Seule la libre disponibilité des sources assure la capacité d'adaptation et de déploiement et surtout la pérennité nécessaires à l'investissement dans une solution logicielle intégrée.

Une autre voie pour appréhender ce que sera un style réflexif de programmation se trouve sans aucun doute du côté des méthodes formelles existantes en informatique : cadres théoriques formels pour la programmation, techniques de preuves de propriétés sur les programmes, techniques de synthèse ou extraction de programmes, systèmes de types permettant l'établissement d'interfaces entre les modules d'un système. En effet, tout métaprogramme non trivial devra être à même de manipuler des programmes non pas simplement sur la forme, leur syntaxe, mais aussi sur le fond, leur *sémantique*. C'est seulement en maîtrisant la sémantique des programmes, en révélant les structures essentielles et en éliminant les complications superflues qu'il sera possible de construire, analyser, communiquer des programmes, et négocier sur des bases solides des contrats entre demandeurs et fournisseurs de services¹⁴ pour interagir

¹⁴ On entend trop souvent une sentence empreinte plus de folie que de sagesse selon laquelle l'expressivité d'un langage serait non dans ce qu'il permet de dire, mais dans ce qu'il interdit de dire. En suivant cette logique, le langage le plus expressif serait celui qui interdit de dire quoi que ce soit ! Notre étude suggère que l'expressivité d'un langage se trouve plutôt dans les contrats qu'il permet de passer, contrats qui peuvent comporter aussi bien des clauses positives que des clauses négatives. Tout au contraire des recommandations «fascistes» des fanatiques de l'interdiction, ou du douteux slogan d'«information hiding» répété par les pontes du développement fermé «orienté objet», il s'agit donc non pas de cacher ou d'interdire de l'information, mais de communiquer et d'utiliser de la métainformation relative à l'utilisation d'autres informations ; cette métainformation, impossible à exprimer formellement dans les cadres de programmation traditionnels fermés et non méta, et pourtant nécessaire à la négociation de contrats adéquats entre acteurs informatisés ou informaticiens.

et coopérer de manière efficace et responsable. On note d'ailleurs que ces acteurs de la vie informatique, demandeurs et fournisseurs de services, peuvent être aussi bien humains que machines ou une intégration des deux ; reste le besoin pour eux de communiquer du sens, et non pas des suites aléatoires de zéros et de uns.

Le sens ici, comme ailleurs, émerge comme épiphénomène des interactions entre acteurs communicants, de leur exploration, superficielle ou profonde, statique ou dynamique, effective ou putative, des structures sémantiques communiquées. D'où encore une fois la nécessité de libre disponibilité des sources, pour que le sens des structures soit pleinement accessible. Cacher les sources des programmes, c'est freiner la communication des structures décrites ; et le développement cloisonné mène en effet à l'utilisation de structures de bas niveau, à la fois sémantiquement pauvres, et lourdes de détails inutiles et encombrants. Un style de programmation réflexif cherchera au contraire, dynamiquement si besoin est, à expliciter autant que possible le sens des concepts échangés, tout en éliminant autant que possible les détails inutiles.

5.3 une nécessité scientifique et industrielle

Nous avons osé parler plus haut de productivité relative offerte par des langages différents. Or, il n'y a pas à notre connaissance d'étude scientifique sérieuse actuellement possible sur le sujet ; l'unité utilisée pour mesurer la productivité est généralement le millier de lignes de code, dont la signification en terme de fonctionnalités varie grandement d'un langage à un autre, et selon le domaine étudié. Pour faire des études scientifiques sérieuses et incontestables, il faudrait avoir une large base de code visible par la communauté scientifique toute entière, qu'il serait possible d'inspecter dans le détail et d'évaluer au vu des fonctionnalités offertes. Mais le principe même des systèmes à développement clos qui dominent l'ensemble de l'industrie est que le code n'est pas publiquement accessible pour une évaluation scientifique ou autre. L'évaluation par les pairs est au contraire l'essence même du développement ouvert [[Raymond, 1997](#)], seule méthode propice à faire progresser l'étude scientifique. Il est aussi évident que pour analyser scientifiquement une quantité grandissante d'information, il faudra utiliser des métaprogrammes plus avancés que des simples compteurs de lignes de code, de lexèmes, ou de tout autre unité syntaxique. Les méthodes réflexives sont donc une nécessité scientifique.

Elles sont aussi une nécessité industrielle. Le développement fermé et non méta fait que l'industrie prend un retard qui se creuse de plus en plus vis-à-vis de la recherche, car elle s'engage constamment les yeux fermés dans des voies de garage avec toute son inertie, en se coupant de tout contrôle rationnel (voir ci-haut §4.2). Seule une évaluation scientifique peut garantir à un industriel, ou à tout un chacun, c'est-à-dire à tout consommateur, que son intérêt est pris en compte par les fournisseurs de services informatiques, développeurs internes ou externes. Nous en avons une illustration flagrante à l'heure où tout le monde s'affole sur le fameux «bogue de l'an 2000», qui nécessite actuellement la correction ou le remplacement *à la main* de millions de lignes de codes, dont les sources sont parfois perdus ! Combien d'efforts auraient été économisés, si les méthodes réflexives avaient été employées lors du développement du code en question ! Non seulement le code, librement disponible donc partagé, aurait été moins long, donc plus simple à remplacer, mais encore sa visibilité aurait grandement facilité la détection des occurrences du bogue, et leur correction complète, plutôt que leur contournement hasardeux ; enfin, la métaprogrammation, et l'usage de langages de haut niveau qu'elle recommande, aurait pu permettre d'automatiser la majeure partie du processus de conversion globale des programmes et des données, à partir d'indications simples et localisées sur les modifications à apporter au traitement des données.

La métaprogrammation et la libre disponibilité des sources reviennent à ouvrir les programmes à tous traitements et modifications, effectués par l'homme comme par la machine ; il s'agit finalement d'une question d'ouverture des programmes à l'intelligibilité, de participation à un processus d'édification d'une

connaissance partagée, c'est-à-dire, de science !

6 Conclusion

Nous pensons avoir démontré, à l'aide d'arguments cybernétiques, que l'informatique a aujourd'hui à faire face à deux défis apparemment différents et cependant intimement liés, la métaprogrammation et la libre disponibilité des sources des logiciels.

Leur avènement commun est inéluctable, car ils sont la clef d'une réduction importante des coûts de production, subis en fin de compte par le consommateur. C'est pour accompagner cet avènement dans de bonnes conditions, pour en tirer le meilleur parti et en éviter les écueils, qu'il est utile de connaître ces deux phénomènes.

Nous ne prétendons pas avoir fait un exposé exhaustif du sujet ; nous espérons seulement en avoir révélé un aspect généralement négligé, et appelons de nos vœux une étude plus poussée du génie logiciel sous l'éclairage d'une théorie de la complexité et de l'expressivité qui prendrait en compte les quelques éléments que nous avons proposés.

Espérons donc que les informaticiens s'affranchissent des slogans clinquants mais creux comme «multi-média», «orienté-objet», «design patterns», «réseau intelligent», etc., sous-produits d'une culture de cloisonnement et de déresponsabilisation. Qu'ils adoptent plutôt une attitude scientifique, c'est-à-dire ouverte et critique à la fois, face aux processus essentiels sous-jacents à leur activité, lesquels, pour être techniques, n'en comportent pas moins des implications économiques, politiques, et somme toute, morales¹⁵.

References

- [Bastiat, 1850] Frédéric Bastiat. Œuvres, 1850. <http://bastiat.org/>. 8
- [Brooks, 1995] Frederick P. Brooks, Jr. *The Mythical Man-Month, Twentieth Anniversary Edition*. Addison-Wesley, 1995. 1
- [Felleisen, 1991] Matthias Felleisen. On the expressive power of programming languages. Technical report, Rice University, 1991. 3.2
- [Kiczales *et al.*, 1997] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. June 1997. 2.3
- [Li and Vitanyi, 1998] Ming Li and Paul Vitanyi. *An Introduction to Kolmogorov Complexity and its Applications, 2nd Edition*. Springer-Verlag, 1998. 3.3
- [Pitrat, 1990] Jacques Pitrat. *Métaconnaissance, futur de l'intelligence artificielle*. Hermès, 1990. 2.1
- [Raymond, 1996] Eric S. Raymond, Editor. The on-line hacker Jargon File, version 4.0.0, 1996. ftp://ftp.gnu.org/pub/gnu/jarg*. 3.2

¹⁵ Toute activité humaine digne de ce nom a de telles implications ; sinon, elle ne susciterait que le désintérêt et personne n'y investirait son temps et autres précieuses ressources. Nul, par ses activités ou ses absences d'activités, en tant que professionnel, amateur, citoyen ou membre d'une famille, n'échappe donc à ses responsabilités d'humain, de vivant, vis à vis des ressources qu'il met en jeu ou qu'il immobilise, directement ou indirectement.

- [Raymond, 1997] Eric S. Raymond. The Cathedral and the Bazaar, 1997.
<http://catb.org/~esr/writings/cathedral.html>. 4.3, 5.3
- [Rideau, 1997] François-René Rideau. Manifeste de la libre information, 1997.
<http://fare.tunes.org/Manifeste.fr.html>. 4.1
- [Shivers, 1996] Olin Shivers. A Universal Scripting Framework or Lambda: the ultimate “little language”.
Concurrency and Parallelism, Programming, Networking, and Security, Lecture Notes in Computer Science, 1179:254–265, 1996. 5.2
- [Turing, 1936] Alan Turing. On computable numbers, with an application to the Entscheidungsproblem.
Proceedings of the London Mathematical Society, 45:161–228, 1936. 3.1
- [Wiener, 1957] Norbert Wiener. *Cybernetics and Society*. Houghton Mifflin, 1957. 3