Dossier de candidature
au concours de
chargé de recherche de deuxième classe du CNRS

Section numéro 7 – Sciences et Technologies de l'Information

Programme de recherche

Applications des méthodes formelles à la certification de logiciels

Jean-Christophe Filliâtre

Domaines d'intérêt

« La programmation est une science exacte », ainsi que l'énonce C. A. R. Hoare dans l'article qui introduit la logique qui porte son nom. D'éminents informaticiens, tels que D. E. Knuth ou E. W. Dijkstra, en ont d'ailleurs fait la preuve avec brio. Cependant, on ne peut que constater que la programmation est considérée en pratique comme une discipline expérimentale, où les tests prédominent sur les raisonnements. Ce raisonnement est pourtant présent dans toute genèse d'un programme, même s'il est parfois informel, et s'il laisse rapidement sa place à une activité expérimentale, c'est parce qu'il n'est pas conservé avec le programme, et encore moins supporté par des outils.

Le but d'une méthode formelle est d'offrir au raisonnement qui sous-tend un programme un véritable statut à côté des lignes de code, une possibilité pour le « pourquoi » d'accompagner le « comment ». Ce qui n'était que de vagues commentaires jusque là, dans le meilleur des cas, devient une véritable spécification mathématique. Ce sont les idées développées à la fin des années 60 par R. W. Floyd et C. A. R. Hoare.

Pour être mises en œuvre, elles nécessitent deux choses : d'une part la définition d'un système logique formel, suffisamment riche pour que l'on puisse y écrire aisément les spécifications d'un logiciel ; et d'autre part des outils de preuve formels permettant de raisonner, interactivement ou automatiquement, à propos de ces spécifications. Le caractère formel de ces outils est rendu nécessaire par la quantité et le caractère très technique des preuves liées à la validation d'un logiciel.

Or, ces deux dernières décennies ont justement vu l'apparition de nombreuses logiques formelles, les assistants de preuve correspondants étant Nqthm, PVS, HOL, Lego, Coq, etc. Ces deux derniers systèmes reposent sur le *Calcul des Constructions Inductives*, une extension du système F de J.-Y. Girard, développée par G. Huet, T. Coquand et C. Paulin.

C'est dans ce cadre-là que je m'intéresse à la certification de programmes.

Travail accompli

La contribution de mon travail de thèse dans ce domaine a été la définition d'une méthode formelle de preuve de programmes impératifs et fonctionnels reposant sur le Calcul des Constructions Inductives comme langage de spécification et le système Coq comme assistant de preuve. Mon travail est dès à présent distribué avec le système Coq, et a été mis en pratique sur quelques algorithmes non triviaux, avec d'excellents résultats.

La méthodologie proposée repose en premier lieu sur une interprétation fonctionnelle des programmes impératifs. Pour définir cette traduction, j'ai combiné une analyse statique des effets et l'utilisation de monades, généralisées pour l'occasion. Si ces deux aspects ont été séparément largement étudiés, leur combinaison semble être originale. Cette traduction a pour principal avantage d'être proche de ce que l'on ferait « à la main », ce qui conduira à des obligations de preuve très naturelles.

Dans un deuxième temps, j'ai proposé une méthode pour établir la correction totale de programmes, dans le cadre général d'un λ-calcul typé d'ordre supérieur. Les programmes sont spécifiés à l'aide de pré- et post-conditions, dans la tradition de la logique de Floyd-Hoare. La formule de correction $\{P\}\ e\ \{Q\}$ est alors interprétée comme la proposition « la traduction fonctionnelle de e réalise la proposition $\forall x.\ P \Rightarrow \exists y.\ Q$ », au sens de la réalisabilité fonctionnelle. Pour établir une telle proposition, on construit un terme de preuve partiel de la proposition $\forall x.\ P \Rightarrow \exists y.\ Q$, bâti sur la traduction fonctionnelle de e. Ce terme de preuve est incomplet en ce qui concerne ses parties logiques, et les parts manquantes constituent les obligations de preuve. J'ai montré la correction de cette méthode, qui repose sur le théorème de réalisabilité, et également sa complétude relative. Si une telle approche permet de retrouver les obligations de preuve de la logique de Floyd-Hoare dans les cas les plus simples, elle a surtout l'avantage de traiter sans difficulté des constructions souvent bien plus délicates, telles que les effets de bords dans les expressions ou les fonctions récursives.

Enfin, j'ai implanté cette méthode dans le système Coq, et l'ai testée sur des programmes non triviaux. Cet aspect pratique était essentiel pour valider la méthode, et il a permis en outre de dégager un certain nombre de problèmes techniques qui ne ressortaient pas de la simple étude théorique, tel que la nécessité d'insérer automatiquement certaines annotations pour n'impliquer l'utilisateur que dans une spécification minimale. J'ai dégagé un grand nombre d'avantages de la méthode que je propose par rapport aux méthodes formelles existantes, tant du point de vue de la spécification que de celui des programmes et des preuves, et cela est notamment dû au fait d'avoir impliqué dans ma démarche des outils de spécification et de preuve existants, à savoir le Calcul des Constructions Inductives et le système Coq.

Perspectives de recherche

Il reste du chemin à parcourir pour parvenir à un environnement complet de preuve de programmes. À côté de certaines améliorations essentiellement techniques (étendre le langage de programmation, par exemple par du filtrage ; fournir une bibliothèque standard de fonctions spécifiées, en particulier autour de la manipulation des nombres à virgule flottante ; tactiques automatiques pour aider à la résolution des obligations de preuve ; etc.), j'entrevois des extensions plus profondes.

Exceptions. Les exceptions constituent une structure de programmation très puissante, qui apporte une solution pour l'instant sans égale à la gestion des erreurs. On trouve des exceptions aussi bien dans les langages impératifs, que fonctionnels ou orientés objets. C'est pour cela qu'il me semble nécessaire d'intégrer à terme des exceptions dans un environnement de développement certifié. J'ai montré dans ma thèse comment les exceptions pouvaient être interprétées dans le cadre de mon travail, en terme d'analyse statique des effets et de traduction fonctionnelle à l'aide de monades. Il reste cependant à définir comment *spécifier* un programme pouvant lever une exception. Une possibilité consiste à étendre la syntaxe d'annotation sous la forme

$$\{P\}\ e\ \{Q + R\}$$

avec la signification usuelle de $\{P\}\ e\ \{Q\}$ lorsque e retourne une valeur normalement, et l'idée que la proposition R est vérifiée dès lors que l'évaluation de e lève une exception — ici dans le cas très simple d'une unique exception sans argument. On est alors en mesure d'associer une propriété à la levée d'une exception et d'utiliser plus tard cette information dans le cas où l'exception est effectivement levée. Ainsi, une fonction *map* de recherche d'une clé k dans une table t pourrait être spécifiée de la manière suivante :

$$\{\}\ (map\ t\ k)\ \{\lambda r.(In\ t\ k\ r) + \forall v.\neg(In\ t\ k\ v)\}$$

Un cas particulier d'usage des exceptions est celui où le programmeur se retrouve dans une situation qu'il sait par ailleurs inaccessible, généralement par contradiction avec une pré-condition, comme dans l'exemple suivant :

$$\begin{array}{l} \{n \neq 0\} \\ \quad \mathsf{match}\ n\ \mathsf{with} \\ \qquad 0 \Rightarrow [\ \textit{code inaccessible}\ ] \\ \quad \mid (S\ p) \Rightarrow p \\ \{r \mid r < n\} \end{array}$$

L'usage est en effet de déclencher une erreur à cet endroit, donc généralement de lever une exception, dont on sait qu'elle ne sera en fait jamais levée. Dès lors, un tel programme pourrait se voir attribuer une post-condition de la forme $\{Q + \mathsf{False}\}$ exprimant que la levée de l'exception est impossible. Mais tout programme au dessus de celui-là déclarera dans ses effets pouvoir potentiellement lever cette exception, qu'il faudra nécessairement prendre en compte dans les obligations de preuve, même s'il s'agira à chaque fois d'une obligation de preuve triviale. Une bien meilleure idée consisterait à introduire un mot clé never que l'on interpréterait comme du code inaccessible et qui serait traduit dans le terme de preuve en une obligation de preuve de l'absurde. Le programme conserverait alors son type initial, contrairement à l'utilisation d'une exception (cette construction pouvant être traduite en une exception lors de la production de code pour un langage particulier).

Modularité. La modularité est un aspect essentiel de la programmation lorsque les développements deviennent importants. Elle permet entre autres à plusieurs développeurs de collaborer aisément, avec pour chacun une connaissance minimale du travail des autres, et de modifier facilement des parties du code avec des répercussions minimales sur le reste du code. La plupart des langages de programmation modernes fournissent une possibilité de programmation modulaire, que se soit à travers un vrai système de modules ou à travers des classes. Une méthode de preuve de programmes se doit donc d'offrir la même possibilité de développement modulaire. C'est d'ailleurs le cas pour la méthode B et le système KIV.

La méthode que j'ai proposée dispose des éléments nécessaires à un développement modulaire de preuve de programmes. En effet, la modularité repose sur l'unique propriété suivante : si la preuve de correction d'un programme e est faite dans un certain contexte Γ contenant $x : \tau$, alors cette preuve ne dépend que du type τ, et cette propriété est immédiate en ce qui concerne ma méthode. Cependant, cela ne définit pas pour autant un système de modules. Pour cela, il faut adopter un système de module particulier pour le langage de programmation et surtout comprendre comment l'étendre pour y ajouter des informations logiques. Il faut tout d'abord donner pour chaque fonction son type *annoté*, c'est-à-dire non seulement son type au sens usuel, mais également ses effets et sa spécification. Mais cela ne suffit toujours pas. En effet, spécifier les fonctions d'un module nécessite également d'exporter avec ces fonctions les définitions permettant de les spécifier. Imaginons par exemple que l'on spécifie un module définissant une table d'association impérative, sous la forme d'un type abstrait t. Alors il faut exporter également un prédicat indiquant si une association appartient ou non à une table donnée, car dans le cas contraire nous serions dans l'impossibilité de spécifier les fonctions opérant sur cette table. Une fois un tel prédicat exporté — appelons-le *In* — alors il est possible de spécifier une fonction telle que celle qui crée une table vide, par exemple de la manière suivante :

$$(\mathit{empty}\ x)\ \{r \mid \forall k.\forall v.\neg(\mathit{In}\ r\ k\ v)\}$$

On remarquera qu'il n'est nul besoin de savoir comment le type t est construit, ni comment le prédicat *In* est défini. L'encapsulation est respectée et permet à tout moment de changer l'implantation de la table d'association. Enfin, il faut pouvoir déclarer également des axiomes dans une interface, tels que par exemple un axiome affirmant qu'une table ne contient au plus qu'une association pour une même clé.

En fin de compte, une interface de module devra comprendre des déclarations au sens usuel, agrémentées d'informations d'effets et de spécifications, ainsi que des déclarations d'objets logiques, qui peuvent être des définitions, des axiomes, etc.

Raffinement. Les méthodes B et KIV proposent la fonctionnalité de *raffinement*, qui permet de raffiner une spécification en une implantation — éventuellement partielle et non-déterministe — ou une implantation en une autre implantation. C'est une très vieille idée, que l'on trouve déjà dans les preuves de programmes développées par C. A. R. Hoare, puis formalisée plus tard par Back et Morgan, qui consiste à construire un programme « de haut en bas » en partant de sa spécification. M. Gordon montre dans son cours d'introduction à la preuve de programmes en logique de Floyd-Hoare comment le raffinement peut être vu de façon très syntaxique comme un ensemble de règles de réécriture sur des formules de la forme $[P, Q]$ dont l'interprétation est « l'ensemble des programmes e tels que $\{P\}\ e\ \{Q\}$ ». Dans notre cas, l'information à donner est le type de calcul annoté κ — regroupant type, effets et spécification — interprété alors comme « n'importe quel programme e de type κ ». Avec un tel formalisme, il est en particulier possible de décrire la structure la plus externe d'un programme sans en donner les parties internes, et d'obtenir tout de suite des obligations de preuve. Ainsi je peux écrire

$$\begin{array}{l}
\{P\} \\
\quad \mathsf{while}\ b\ \mathsf{do} \\
\quad\quad \{\mathsf{invariant}\ I\} \\
\quad\quad [c : \kappa] \\
\quad \mathsf{done} \\
\{Q\}
\end{array}$$

où P, b, I et Q sont donnés, mais pas le corps de la boucle c, dont on donne uniquement le type de calcul κ, de la forme $(r : \tau, \epsilon, P_c, Q_c)$. On peut imaginer dans le cas présent que P_c et Q_c prennent par défaut la valeur de I, exprimant ainsi la préservation de l'invariant par le corps de la boucle, mais ce n'est en rien une obligation. On obtient alors en particulier les obligations de preuve $P \Rightarrow I$ et $I \wedge b = \mathsf{false} \Rightarrow Q$, que l'on peut s'attacher à prouver sans plus tarder. On obtient également des obligations de la forme $I \wedge b = \mathsf{true} \Rightarrow P_c$ et $Q_c \Rightarrow I$ dans le cas où la spécification κ de c n'exprime pas directement la préservation de l'invariant. Ainsi on vérifie immédiatement la pertinence de l'invariant I, avant de chercher à en prouver la préservation. C'est là l'intérêt du raffinement, car il permet ainsi de trouver rapidement les erreurs dans la spécification et dans le code.

Il s'avère que l'on peut déjà procéder par raffinement dans le système que je propose, de manière « artisanale ». En effet, on peut déclarer un programme c de type κ et écrire la portion de code ci-dessus avec c comme corps de la boucle. Dès lors, on obtient bien les obligations de preuve recherchées, et une fois ces obligations de preuve établies, on peut remplacer c par un morceau de programme, qui peut contenir à son tour des variables de programme, et ainsi de suite. Bien entendu, un véritable mécanisme de raffinement doit proposer d'autres fonctionnalités, telles que la possibilité d'affaiblir une post-condition ou de renforcer une pré-condition. Mais on peut dès à présent constater que le raffinement est envisageable dans mon système, et il est intéressant de noter qu'il l'est *pour la même raison que la modularité est elle-même possible* : en effet, c'est le fait qu'une preuve de correction ne dépende que du *type* des sous-programmes qui rend possible le raffinement, tout comme la modularité.

Avant d'être des aspects théoriques, la modularité et le raffinement sont surtout des aspects *méthodologiques*. La puissance d'une méthodologie telle que la méthode B ne réside pas dans celle des programmes que l'on peut y décrire, ou dans ses aspects méta-théoriques, mais dans la méthodologie de certification qu'elle propose, s'adaptant ainsi à un large public d'horizons divers. C'est dans cette optique qu'il faudra concevoir les développements futurs de la méthode que je propose, car négliger les aspects méthodologiques, c'est vouer une recherche scientifique à ne jamais dépasser le stade de la recherche elle-même.