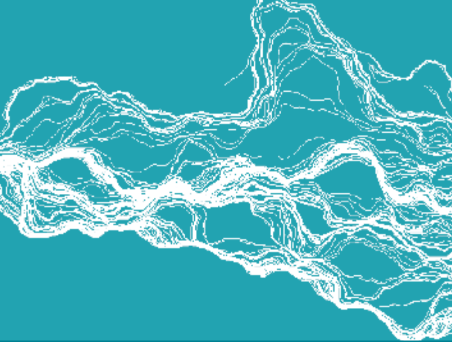

les cahiers de
L'INRIA
Rendez-vous avec les sciences et technologies de l'information et de la communication

MÉTHODES FORMELLES

Comment faire confiance à un compilateur ?

Traduttore, tradittore (traducteur, traître), dit l'adage italien. Si l'on accepte le parallèle entre traduction et compilation de programme, celle-ci consistant à traduire un langage évolué en langage machine, mieux vaut faire mentir l'adage ! Portrait du premier compilateur en voie d'être garanti « zéro faute ».

Comment faire exécuter par une machine des programmes écrits dans un langage de haut niveau[(1)], qu'elle ne peut comprendre ? La manière la plus efficace, appelée compilation, consiste à traduire automatiquement ce « langage source », compréhensible par les programmeurs, en langage exécutable par l'ordinateur (langage cible). Cette traduction est réalisée par des programmes spécialisés appelés compilateurs. La tâche est difficile en raison de l'énorme différence d'expressivité - on parle de « fossé sémantique » - entre langages source et cible. Toutes proportions gardées, il s'agit de traduire *À la recherche du temps perdu* dans les mimiques et grognements de la langue imaginaire du film La guerre du feu ! Autant dire que les sources d'erreurs potentielles sont nombreuses. L'objectif du projet CompCert*, qui associe plusieurs équipes françaises, est de concevoir un compilateur entièrement vérifié.

Idée reçue : au cinéma, on voit souvent des informaticiens scruter un écran rempli de 0 et de 1 et y reconnaître au premier coup d'œil le virus que des méchants ont introduit dans le système. La réalité est bien différente. S'il est vrai que les microprocesseurs ne comprennent qu'un langage très primitif exprimé sous forme de nombres, les informaticiens, pour leur part, écrivent et lisent les programmes dans des langages de programmation de beaucoup plus haut niveau, représentés sous forme de textes et offrant des constructions évoluées pour exprimer l'enchaînement des calculs, décrire les structures de données et décomposer les programmes en morceaux compréhensibles indépendamment. Ces langages de haut niveau constituent le principal vecteur de la pensée informatique[(1)].

La compilation ne se limite toutefois pas à la traduction en langage de la machine de ces constructions en langage de haut niveau : les compilateurs modernes cherchent aussi à améliorer les performances du code machine produit, en appliquant des techniques dites d'optimisation du code. Même si la compilation est un domaine bien maîtrisé de la science informatique[(2)], les compilateurs sont donc des programmes particulièrement complexes. Difficile, dès lors, d'être certain de la justesse de la traduction, c'est-à-dire de l'absence de bugs. Or de même qu'un contresens dans une traduction entre langues naturelles change le sens d'un texte, un bug dans un compilateur peut le conduire à produire, à l'insu du programmeur, un code machine ne respectant pas du tout ses intentions telles qu'exprimées dans le programme source.

Un exemple de tel bug a récemment été découvert dans le système d'exploitation Linux[(3)] : une optimisation astucieuse du compilateur GCC a supprimé un test nécessaire à la sécurité du système, ouvrant la voix à une attaque. Les effets peuvent devenir catastrophiques, ce qui n'était pas le cas de cet exemple, si les logiciels concernés contrôlent les commandes électroniques de vol d'un avion, ou bien le fonctionnement d'un réacteur nucléaire, d'un pacemaker, d'une machine de radiothérapie, etc. : autant de logiciels critiques où toute erreur met en danger des vies humaines. Les procédés de production d'un logiciel critique sont de fait beaucoup plus rigoureux que ceux de Linux ou de tout autre logiciel grand public.

Pour être qualifié comme utilisable dans un environnement critique, un tel logiciel doit passer des tests extrêmement poussés et avoir été développé suivant les meilleures pratiques connues. En particulier, les optimisations automatiques sont exclues et le code assembleur produit par le compilateur est examiné minutieusement (à la main ou avec l'aide d'outils informatiques). Mais si ces pratiques diminuent les risques d'erreur, elles ne les éliminent pas entièrement, outre le fait qu'elles sont de plus en plus coûteuses et ne sont guère adaptées à de gros logiciels. D'où l'entrée en scène des méthodes dites formelles, dans l'industrie du logiciel critique.

Les tests classiques démontrent en effet la présence de bugs mais jamais leur absence, selon le dicton du célèbre informaticien néerlandais Edsger W. Dijkstra (1930-2002). À l'inverse, les méthodes formelles construisent un modèle mathématique du comportement du programme et établissent, par le calcul et la déduction logique, des propriétés vraies pour toutes les exécutions possibles du programme. Parmi ces méthodes, sur lesquelles il est inutile de s'appesantir ici : l'analyse statique, la vérification par modèles ou la preuve de programme. Les propriétés recherchées vont de l'absence de « plantages » à l'exécution (pas de division par zéro, de débordements arithmétiques, d'accès mémoire invalides) jusqu'à la conformité complète du programme à ses spécifications. Par exemple, Airbus utilise en production des outils de vérification formelle comme les analyseurs statiques Astrée[4], aiT WCET[5] et Caveat[6].

Le passage dans la pratique des méthodes formelles, concrétisant trente ans de recherches longtemps restées théoriques, est l'un des grands succès récents de la science informatique. Notons cependant que, jusqu'à présent, les outils de vérification formelle travaillaient généralement sur le programme source (donc en langage évolué) et non pas sur le code exécutable. Par conséquent, une compilation incorrecte pouvait là encore invalider les garanties obtenues par vérification formelle du code source. Le compilateur apparaît donc de plus en plus comme un des maillons faibles dans la chaîne de production du logiciel critique, ce qui nous a conduits à l'idée d'appliquer les méthodes formelles au compilateur lui-même.

L'enjeu est alors de prouver mathématiquement un théorème de préservation sémantique (préservation du sens, de l'enchaînement des calculs, etc.) pour le compilateur. Énoncé du théorème : si aucune erreur n'est signalée à la compilation, alors le code machine produit se comporte exactement comme le prédit la sémantique du programme source. Un tel théorème fournit des garanties sans commune mesure avec celles obtenues par les tests classiques, car elles sont vraies pour tous les programmes sources. Comme corollaire, on obtient que toute propriété comportementale vraie à propos du programme source s'étend automatiquement au code exécutable, ce qui justifie du coup pleinement l'application de méthodes formelles sur les programmes sources.

Prouver mathématiquement un compilateur n'est pas une idée nouvelle : les premières publications sur ce sujet remontent à 1967 (preuve sur papier)[7] et 1972 (preuve vérifiée par machine)[8]. Mais il s'agissait d'un compilateur assez trivial opérant sur un tout petit langage d'expressions arithmétiques. Depuis, de nombreux travaux de recherche ont fait progresser le domaine à un point tel que l'on peut désormais envisager la preuve d'un compilateur réaliste, utilisable pour des logiciels embarqués critiques. Tel est l'objectif de notre projet CompCert[9].

© COMPUTER HISTORY MUSEAUM

Au début des années 1950, l'américaine Grace Hopper (1906-1992) et son équipe réalisent pour la première fois un programme capable de traduire des commandes écrites en anglais dans un langage machine. Cette mathématicienne, amirale de l'armée américaine, est considérée comme l'inventeuse du compilateur.

Le compilateur vérifié CompCert traduit le langage Clight, un grand sous-ensemble du langage C couramment utilisé pour l'écriture de logiciels critiques, en code assembleur pour les processeurs PowerPC et ARM (deux microprocesseurs très utilisés dans les systèmes embarqués critiques). Il effectue quelques optimisations afin de produire du code suffisamment rapide et suffisamment compact pour les applications visées. L'architecture du compilateur CompCert est très classique (voir le schéma page suivante) : il se décompose en une douzaine d'étapes exécutées de manière séquentielle, dites « passes de compilation », chacune effectuant une transformation bien identifiée. Ces passes communiquent *via* des langages intermédiaires de niveau sémantique de plus en plus bas : pour utiliser une métaphore, ce sont autant de marches taillées dans la falaise qui sépare langage source et langage machine.

La grande nouveauté du compilateur CompCert est que chaque passe de compilation est accompagnée d'une preuve mathématique : la preuve que la sémantique du programme est préservée. La composition de ces passes, c'est-à-dire le compilateur entier, la préserve donc également. Pour énoncer et prouver cette propriété, il est évidemment nécessaire de définir mathématiquement la sémantique de chacun des langages impliqués : le langage source (le langage Clight), le langage cible (l'assembleur des processeurs PowerPC et ARM), mais aussi chacun des huit langages intermédiaires.

Ces sémantiques définissent exactement les comportements possibles d'un programme : est-ce qu'il termine normalement

* le projet CompCert, coordonné par l'Inria, associe les équipes-projets Gallium et Marelle de l'institut, le laboratoire Cédric commun au Conservatoire national des arts et métiers (Cnam) et à l'École nationale supérieure d'informatique pour l'industrie et l'entreprise (ENSIIE), et le laboratoire Preuves, programmes et systèmes (PPS), unité mixte CNRS-université Paris Diderot.

* pour en savoir plus sur l'assistant de preuve Coq, voir http : //coq.inria.fr/.

INRIA

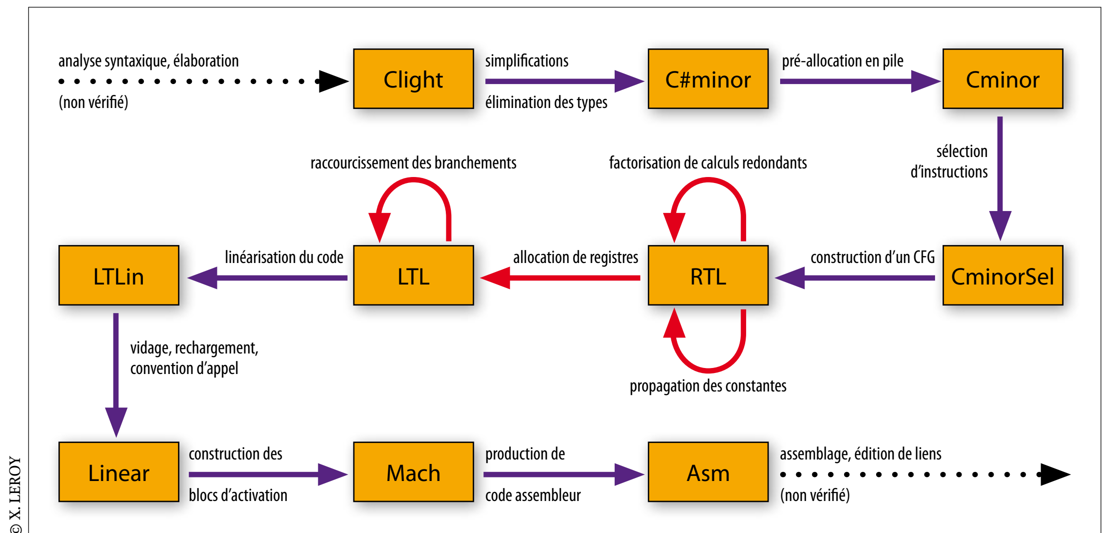

Schéma d'ensemble du compilateur vérifié CompCert : chaque boîte représente un des langages impliqués dans la compilation, depuis le langage source Clight (en haut à gauche) jusqu'au langage assembleur cible (en bas à droite). Chaque flèche représente une étape, ou « passe de compilation ». Certaines passes (flèches rouges) sont des optimisations qui éliminent des inefficacités présentes dans le programme et tirent parti des possibilités spécifiques du processeur. Par exemple, la passe de factorisation des calculs redondants réutilise les résultats de calculs effectués antérieurement, au lieu de refaire ces calculs. Les autres passes sont de simples traductions qui éliminent des traits de haut niveau en les ré-exprimant avec des traits de plus bas niveau. Au final, ce compilateur produit du code exécutable dont les performances rivalisent avec celles du code produit par un compilateur de référence (GCC au premier niveau d'optimisation).

sans erreur ? Ou bien termine-t-il prématurément sur une erreur (plantage) ? Ou encore s'exécute-t-il à l'infini (divergence) ? Dans les trois cas, la sémantique associe au programme une trace complète des opérations d'entrée-sortie qu'il effectue : lecture du clavier, affichage à l'écran, consultation d'un capteur, envoi d'un ordre à un actionneur, etc. Le résultat de préservation sémantique dit que ces traces sont préservées et donc que les programmes avant et après compilation se comportent exactement de la même manière pour un regard extérieur. Problème : écrire la preuve de ce résultat à la main et sur le papier conduirait à remplir des milliers de pages et, qui plus est, aucun mathématicien n'accepterait de les relire et de les vérifier.

Nous nous sommes donc appuyés sur l'assistant de preuve Coq*, développé par l'Inria depuis une vingtaine d'années. Ce logiciel permet en effet d'écrire des spécifications et d'énoncer des théorèmes dans un langage proche des mathématiques, puis de prouver ces théorèmes, en interaction avec l'utilisateur, et enfin de vérifier ces preuves *a posteriori*. Il ne s'agit toutefois pas de démonstration automatique : seule la vérification finale s'effectue sans intervention humaine. La construction de la preuve est en revanche guidée par l'utilisateur et bien des propriétés évidentes pour un humain demandent de longues explications détaillées pour quelles soient transposables à la machine. La construction d'une preuve comme celle du compilateur CompCert représente donc un effort considérable : environ 48 000 lignes de Coq (à comparer aux 8 000 lignes de programme qui constituent le compilateur lui-même) et près de quatre ans de travail pour une personne (à comparer aux trois mois nécessaires pour écrire le compilateur seul).

Résultat : une certitude sans précédent dans la validité de la preuve, certitude bien supérieure à celle qu'offrirait une preuve traditionnelle sur le papier. Une partie du chemin reste toutefois à parcourir avant qu'un compilateur formellement vérifié comme CompCert n'entre dans la boîte à outils du développeur de logiciels critiques. Par exemple : quelques éléments du compilateur n'ont pas encore fait l'objet d'une preuve (l'analyseur syntaxique, en amont, et l'assemblage et l'édition de liens, en aval), de nombreuses optimisations une fois prouvées pourraient être ajoutées… On pourrait également envisager d'autres langages sources que le langage C. Enfin, les preuves menées jusqu'ici ne s'appliquent qu'à des programmes séquentiels. Les étendre aux programmes parallèles, nécessaires pour tirer pleinement parti des processeurs « multicœurs » modernes, pose de formidables problèmes de sémantique des langages et des processeurs.

Les résultats d'ores et déjà obtenus n'en établissent pas moins que, oui, il est à notre portée de vérifier formellement un compilateur réaliste. De plus, les techniques et outils de compilation, de sémantique et de preuve sur machine disponibles aujourd'hui sont suffisamment puissants pour mener à bien une telle vérification, même si de nombreuses améliorations restent souhaitables. Nous pouvons désormais imaginer de généraliser la vérification formelle à d'autres outils intervenant dans la production et la qualification des logiciels critiques, au-delà du seul compilateur C : aux générateurs automatiques de code, aux analyseurs statiques, aux vérificateurs de programmes, etc. Le but, à terme, est bien sûr d'obtenir des environnements de développement dignes de la plus haute confiance pour la production des logiciels critiques.

Xavier Leroy, directeur de recherche Inria, est responsable de l'équipe-projet Gallium. Il est l'un des principaux auteurs du langage Caml. Ses recherches portent sur l'amélioration de la sûreté et de la sécurité des logiciels.

[1] G. Berry. *Pourquoi et comment le monde devient numérique*. Fayard, 2008
[2] A. W. Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, 1998.
[3] Clever attack exploits fully-patched Linux kernel. *The Register*, http : //www.theregister.co.uk/2009/07/17/, juillet 2009.
[4] B. Blanchet et *al., in* Programming Language Design and Implementation 2003, pp. 196-207. ACM Press, 2003.
[5] Ch. Ferdinand et *al., in Embedded Software 2001*, pp. 469-485. LNCS 2211, Springer, 2001
[6] P. Baudin et *al., in International Conference on Dependable Systems and Networks* (DSN 2002), pp. 537-537. IEEE Computer Society Press, 2002
[7] J. McCarthy *and* J. Painter, *in Mathematical Aspects of Computer Science*, vol. 19 of *Proc. of Symposia in Applied Mathematics*, pp 33–41. American Mathematical Society, 1967.
[8] R. Milner *and* R. Weyhrauch, *in, Proc. 7 th Annual Machine Intelligence Workshop* (B. Meltzer *and* D. Michie, ed.), volume 7 of *Machine Intelligence*, pp. 51-72. Edinburgh University Press, 1972.
[9] X. Leroy, *Communications of the ACM*, 52 (7), pp.107-115, 2009.