DOI:10.3166/TSI.26.1195-1200

Comment gagner confiance en C ?

Le langage C est très utilisé dans l'industrie, en particulier pour développer du logiciel embarqué. Un des intérêts de ce langage est que le programmeur contrôle les ressources nécessaires à l'exécution des programmes (par exemple, la géographie de la mémoire, ou encore les allocations et libérations de cellules de la mémoire), qui de plus influent sur les performances des programmes. Des programmes C peuvent ainsi être particulièrement efficaces, mais le prix à payer est un effort de programmation. Par exemple, il peut être nécessaire d'utiliser l'arithmétique de pointeurs afin de calculer l'adresse d'une cellule de la mémoire.

Cependant, le fait que le langage C laisse davantage de liberté au programmeur favorise également la présence d'erreurs à l'exécution des programmes, erreurs qui peuvent être difficiles à détecter. Le dépassement des bornes de tableaux, ou encore la non-initialisation de variables sont des exemples de telles erreurs, qui peuvent ne pas se produire dans des langages plus récents tels que Java. D'autres erreurs à l'exécution proviennent de conversions de types et plus généralement de l'absence de sûreté du typage pour C.

Des solutions existent pour assurer sur du logiciel C des résultats de sûreté comparables à ceux garantis par les compilateurs d'autres langages plus récents. De nombreux systèmes effectuent des analyses statiques afin de détecter des erreurs dans les programmes C, principalement en vérifiant la sûreté des accès à la mémoire. Par exemple, l'analyse statique de Ccured repose sur un système de types qui étend celui de C et permet de séparer les pointeurs selon leur usage de la mémoire (Necula *et al.*, 2002). Ainsi, un programme C transformé par Ccured comprend soit des accès sûrs à la mémoire, soit des accès à la mémoire dont la sûreté n'est pas garantie par l'analyse statique, et pour lesquels des assertions devant être vérifiées à l'exécution des programmes sont insérées. D'autres systèmes tels que Cyclone proposent des dialectes plus sûrs du langage C empêchant des erreurs telles que celles détectées par Ccured de se produire (Jim *et al.*, 2002).

Plus généralement, vérifier formellement un programme consiste à le spécifier au moyen d'annotations écrites sous forme de formules logiques et à établir ensuite que le programme satisfait sa spécification, c'est-à-dire que la spécification et le programme respectent les règles d'une logique de Hoare définissant les exécutions valides de chaque instruction du langage source. Les assertions permettent d'exprimer des propriétés attendues du programme. Les assertions qu'il est nécessaire d'écrire dans les programmes sont les pré et post-conditions des fonctions, les invariants de boucle

et éventuellement les conditions de terminaison des boucles. La vérification formelle d'un programme produit des obligations de preuve qui sont en général déchargées vers des outils externes d'aide à la preuve. Par exemple, le vérificateur de programmes C Caduceus produit des obligations de preuve qui soit sont prises en charge par des procédures de décision, soit nécessitent d'être vérifiées à l'aide d'un assistant à la preuve (Filliâtre *et al.*, 2004).

Récemment, des extensions de la logique de Hoare adaptées aux langages impératifs manipulant des pointeurs ont été proposées. Ainsi, la logique de séparation définit un langage d'assertions permettant d'observer finement la mémoire et donc de décrire aisément des propriétés usuellement recherchées sur des pointeurs (O'Hearn *et al.*, 2001) : par exemple, le non-chevauchement (*i.e.* la séparation) entre zones de la mémoire, ou encore l'absence de cycle dans une structure de données chaînée. En logique de séparation, une propriété relative à une instruction ne concerne que l'empreinte mémoire de l'instruction, c'est-à-dire les zones de la mémoire utilisées lors de l'exécution de l'instruction. Contrairement à la logique de Hoare, la logique de séparation permet de raisonner localement sur l'empreinte mémoire d'une instruction, et garantit que les autres portions de la mémoire ne sont pas modifiées par l'exécution de cette instruction, ce qui facilite grandement la vérification des programmes. La logique de séparation commence également à être utilisée afin d'effectuer des analyses statiques sophistiquées de pointeurs. Actuellement, les outils automatisant le raisonnement en logique de séparation opèrent sur de petits langages impératifs (Berdine *et al.*, 2005).

Ainsi, il existe de nombreuses solutions pour éviter des erreurs dans les programmes C. Mais, une fois certaines propriétés vérifiées sur un code écrit en C, comment garantir qu'elles sont également vérifiées par le code cible produit par le compilateur C ? Des bogues dans le compilateur peuvent invalider la vérification du code source. Il est alors nécessaire de vérifier formellement une propriété exprimant l'équivalence entre un code source et le code compilé correspondant. Différentes équivalences sont envisageables et sont plus ou moins difficiles à établir. La propriété équivalente peut être une transcription en langage cible de la propriété ayant déjà été prouvée sur le programme source. Par exemple, il peut s'agir de vérifier la sûreté des accès à la mémoire du programme cible, ou encore que le programme cible satisfait la spécification fonctionnelle que le programme source satisfait également. Une autre façon de vérifier que deux programmes sont équivalents consiste à vérifier qu'ils effectuent les mêmes calculs observables. L'exécution d'un programme est ainsi abstraite en le calcul des entrées-sorties et des valeurs finales du programme.

La validation de traducteurs est une première solution pour vérifier formellement les résultats d'un compilateur (Pnueli *et al.*, 1998). Dans cette approche, le compilateur est considéré comme une boîte noire traduisant un programme source en un programme cible. Un vérificateur indépendant supposé certifié vérifie *a posteriori* que chaque programme compilé calcule les mêmes résultats que le programme source ayant servi à l'engendrer. La comparaison des résultats repose par exemple sur une

exécution symbolique des deux programmes et sur une comparaison des états correspondants en certains points d'observation des programmes.

Une autre approche est celle du compilateur certifiant, qui produit un code autocertifié « proof-carrying code ») contenant un code compilé ainsi qu'une propriété exprimant que ce code compilé préserve la sémantique du code source correspondant (Necula, 1997). Cette propriété est ensuite prouvée de façon indépendante, établissant ainsi la correction de la compilation du programme initial. Ces deux approches ont été utilisées conjointement, mais pour des langages de bas niveau tels que par exemple les langages d'assemblage typés.

Plutôt que vérifier chaque programme à compiler, une solution consiste à vérifier formellement le compilateur, une fois pour toutes. On obtient ainsi un compilateur certifié (Leroy, 2006b). Un compilateur effectue une succession de passes de transformation de programmes. Une transformation désigne soit une traduction vers un langage de plus bas niveau, soit une optimisation de programmes. Vérifier formellement un compilateur consiste à vérifier formellement chacune des passes.

Cette chronique est issue de notre expérience dans le projet Compcert qui a pour objectif de vérifier à l'aide de l'assistant à la preuve Coq un compilateur réaliste utilisable pour le logiciel embarqué critique. Il s'agit d'un compilateur d'un large sous-ensemble du langage C (dans lequel il n'y a pas d'instructions de sauts) qui produit du code assembleur PowerPC et effectue diverses optimisations (principalement, propagation de constantes, élimination des sous-expressions communes et réordonnancement d'instructions) afin de produire du code raisonnablement compact et efficace (Leroy, 2006b; Blazy *et al.*, 2006). Le compilateur dispose de six langages intermédiaires en plus des langages source et cible. Une sémantique formelle est définie pour chacun des huit langages du compilateur. Elle repose sur un modèle mémoire commun à tous les langages du compilateur (Blazy *et al.*, 2005). La sémantique formelle du langage source impose des accès sûrs à la mémoire, principalement en lecture et écriture.

Toutes les parties vérifiées du compilateur sont programmées directement dans le langage de spécification de Coq, en style fonctionnel pur. Le mécanisme d'extraction de Coq produit automatiquement le code Caml du compilateur à partir des spécifications. Ce code constitue le compilateur dit certifié. Vérifier le compilateur consiste à établir en Coq le théorème de préservation sémantique suivant : pour tout code source S, si le compilateur transforme S en le code machine C sans signaler d'erreur de compilation, et si S a une sémantique bien définie, alors C a la même sémantique que S, à équivalence observationnnelle près. Ainsi, les deux codes sont considérés comme ayant la même sémantique s'il n'y a pas de différence observable entre leur exécution.

D'une façon générale, lors de la vérification d'un compilateur, les événements observés durant l'exécution d'un programme C peuvent se limiter aux résultats du programme. D'autres événements peuvent également être observés, en fonction de la précision de la sémantique choisie. Par exemple, le graphe des appels de fonctions, ou encore la trace des accès à la mémoire en lecture et écriture peuvent être observés en

plus des résultats du programme. La confiance dans le compilateur sera d'autant plus grande que les événements observés sont nombreux et variés.

Le style de sémantique choisi a un impact fort sur les résultats de préservation sémantique pouvant être vérifiés. Il existe principalement deux grandes familles de sémantique opérationnelle adaptées à la preuve d'équivalence de programmes. Une sémantique (naturelle) à grands pas relie un programme à son résultat final. Une sémantique à petits pas détaille tous les états intermédiaires de l'exécution d'un programme, permettant des observations plus précises et rendant compte des exécutions qui ne terminent pas. Les sémantiques à grands pas étant plus simples, elles sont donc souvent choisies afin de raisonner sur des programmes écrits dans des langages tels que C. Cependant, elles ne permettent pas d'observer des programmes dont l'exécution ne termine pas. Les sémantiques à petits pas permettent d'établir des résultats d'équivalence plus forts. Par contre, elles exposent trop d'étapes de calculs, ce qui peut compliquer énormément les preuves. En effet, la preuve d'équivalence sémantique entre un programme p et un programme transformé p_t est rendue difficile par le fait qu'un état intermédiaire de l'exécution de p n'a pas nécessairement d'équivalent dans l'exécution de p_t.

Comment choisir un style de sémantique et valider ensuite une sémantique ? Le principal critère de choix d'une sémantique est la facilité à raisonner sur cette sémantique. Selon le choix, le principe d'induction associé à une sémantique sera plus ou moins difficile à formuler, et les étapes de preuves associées seront plus ou moins difficiles à établir. C'est la diversité des propriétés sémantiques ayant été prouvées qui atteste de la validité de cette sémantique. Vérifier formellement un compilateur optimisant est donc aussi l'occasion de valider la sémantique des langages du compilateur, et en particulier la sémantique de son langage source.

Un résultat intéressant du projet Compcert est que la structure générale du compilateur est conditionnée non pas par les transformations de programmes, mais très fortement par le choix des langages utilisés dans le compilateur, et aussi par le style de sémantique donné à ces langages. Ainsi, les langages intermédiaires du compilateur ont été conçus dans le but de faciliter les preuves de traduction. Souvent, lorsqu'une preuve de traduction d'un langage L_1 en un langage L_2 a nécessité de modéliser différents concepts (par exemple la mise en correspondance de zones de la mémoire), rendant ainsi ces preuves difficiles à faire et à maintenir, un langage intermédiaire L_i entre L_1 et L_2 a été défini. Vérifier séparément la correction des deux traductions vers et depuis L_i s'est avéré beaucoup plus aisé que vérifier la correction de la traduction de L_1 vers L_2. C'est pourquoi le compilateur Compcert dispose de six langages intermédiaires.

Dans Compcert, le style sémantique à grands pas a d'abord été choisi pour son confort. Les sémantiques à grands pas du compilateur certifié permettent d'observer les valeurs finales d'un programme, ainsi que la trace des appels de fonctions effectués durant l'exécution du programme. Il est ainsi vérifié qu'un code compilé calcule les mêmes résultats et exécute aussi les mêmes instructions d'entrées et sorties (c'est-à-

dire les mêmes appels aux fonctions d'entrées et sorties) que le code source ayant servi à l'engendrer.

Une fois les langages intermédiaires définis et les transformations de programmes vérifiées, d'autres styles de sémantique ont été expérimentés afin d'étendre les propriétés vérifiées aux programmes dont l'exécution ne termine pas. L'équivalence entre ces styles de sémantique et les sémantiques à grands pas a été vérifiée, fournissant ainsi une validation supplémentaire des sémantiques étudiées. Ainsi, des sémantiques à petits pas ont été définies pour les langages de bas niveau du compilateur, et les preuves de correction des transformations de programmes associées ont été adaptées sans trop de difficultés. Des expériences récentes de définitions en Coq de sémantiques coinductives pour de petits langages et la vérification en Coq de leur équivalence avec des sémantiques à petits pas pour des programmes qui divergent semblent prometteuses pour les langages de haut niveau du compilateur (Leroy, 2006a).

Des sémantiques axiomatiques ont également été définies afin de relier compilation certifiée et preuve de programmes (Appel *et al.*, 2007). A l'avenir, il est en effet envisagé de vérifier en plus des propriétés garanties par le compilateur certifié, des propriétés définies par le programmeur et exprimées en logique de séparation. Un plongement profond en Coq de la logique de séparation a été défini pour Cminor, un langage intermédiaire du compilateur proche de C. La vérification de la correction de cette sémantique axiomatique par rapport aux sémantiques précédentes augmente à nouveau la confiance en chacune des sémantiques.

En conclusion, diverses solutions existent pour produire du logiciel C de confiance. La vérification de programmes C prend en charge plus ou moins automatiquement des propriétés spécifiées sous la forme d'assertions dans les programmes. La compilation certifiée garantit que tout code compilé se comporte comme prescrit par la sémantique du code source. La définition de sémantiques formelles adaptées au raisonnement sur machine est au centre de la compilation certifiée. Dans un compilateur certifié, de telles sémantiques garantissent de plus des propriétés supplémentaires du langage C, telle la sûreté de certains accès à la mémoire. L'avenir de la vérification de programmes C est dans l'utilisation conjointe de ces solutions.

Bibliographie

Appel A. W., Blazy S., « Separation logic for small-step Cminor », *Theorem Proving in Higher Order Logics, 20th Int. Conf. TPHOLs 2007*, vol. 4732 of *Lecture Notes in Computer Science*, Springer, p. 5-21, 2007.

Berdine J., Calcagno C., O'Hearn P. W., « Smallfoot : Modular Automatic Assertion Checking with Separation Logic. », F. S. de Boer, M. M. Bonsangue, S. Graf, W. P. de Roever (eds), *FMCO*, vol. 4111 of *Lecture Notes in Computer Science*, Springer, p. 115-137, 2005.

Blazy S., Dargaye Z., Leroy X., « Formal Verification of a C Compiler Front-End. », J. Misra, T. Nipkow, E. Sekerinski (eds), *symposium on Formal Methods, 14th FM'07*, vol. 4085 of *Lecture Notes in Computer Science*, Springer, p. 460-475, 2006.

Blazy S., Leroy X., « Formal Verification of a Memory Model for C-Like Imperative Languages. », K.-K. Lau, R. Banach (eds), *ICFEM*, vol. 3785 of *Lecture Notes in Computer Science*, Springer, p. 280-299, 2005.

Filliâtre J.-C., Marché C., « Multi-prover Verification of C Programs. », J. Davies, W. Schulte, M. Barnett (eds), *ICFEM*, vol. 3308 of *Lecture Notes in Computer Science*, Springer, p. 15-29, 2004.

Jim T., Morrisett J. G., Grossman D., Hicks M. W., Cheney J., Wang Y., « Cyclone : A Safe Dialect of C. », C. S. Ellis (ed.), *USENIX Annual Technical Conference, General Track*, USENIX, p. 275-288, 2002.

Leroy X., « Coinductive Big-Step Operational Semantics. », P. Sestoft (ed.), *ESOP*, vol. 3924 of *Lecture Notes in Computer Science*, Springer, p. 54-68, 2006a.

Leroy X., « Formal certification of a compiler back-end or : programming a compiler with a proof assistant. », J. G. Morrisett, S. L. P. Jones (eds), *POPL*, ACM, p. 42-54, 2006b.

Necula G. C., « Proof-Carrying Code. », *POPL '97 : 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, p. 106-119, 1997.

Necula G. C., McPeak S., Weimer W., « CCured : type-safe retrofitting of legacy code », *POPL '02 : Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, ACM Press, New York, NY, USA, p. 128-139, 2002.

O'Hearn P. W., Reynolds J. C., Yang H., « Local Reasoning about Programs that Alter Data Structures. », L. Fribourg (ed.), *CSL*, vol. 2142 of *Lecture Notes in Computer Science*, Springer, p. 1-19, 2001.

Pnueli A., Siegel M., Singerman E., « Translation Validation. », B. Steffen (ed.), *TACAS*, vol. 1384 of *Lecture Notes in Computer Science*, Springer, p. 151-166, 1998.

Sandrine Blazy
ENSIIE - CEDRIC
Sandrine.Blazy@ensiie.fr