Proposition d'ARC INRIA 2003

Concert : "Compilateurs Certifiés"

Cristal, Foc-Cnam, Lemme, Mimosa, Miró, Oasis

23 novembre 2002

Résumé

Concert a comme objectif principal de déterminer s'il est faisable, dans l'état actuel des connaissances, de réaliser un compilateur réaliste qui soit certifié, c'est-à-dire accompagné d'une preuve Coq d'équivalence sémantique entre le code source et le code machine engendré.

Nous proposons de réunir dans cette proposition les chercheurs intéressés dans la production de compilateurs certifiés pour des langages variés, mais travaillant à un niveau intermédiaire entre les langages de très haut niveau (langages fonctionnels ou langages à objet) et langages de bas niveau comme l'assembleur. Le langage visé devrait correspondre à un sous-ensemble de C, éventuellement avec des traits à objets ou fonctionnels.

Cette ARC est l'occasion de regrouper les compétences et expériences des participants en compilation optimisante, sémantique formelle, formalisation de langages machines, et preuves en Coq.

1 Participants à l'ARC

- Cristal, INRIA Rocquencourt: Xavier Leroy, éventuellement un thésard à partir de mi-2003.
- équipe CPR (Cedric-CNAM): Catherine Dubois [1], Sandrine Blazy [2], Olivier Boîte [3] (thésard, actuellement sur une bourse du ministère).
- Lemme, INRIA Sophia: Yves Bertot (coordinateur), Kuntal Das Barman (thésard, actuellement sur Vérificard).
- Mimosa, INRIA Sophia: Manuel Serrano.
- Miró, INRIA Lorraine & INRIA Sophia: Joëlle Despeyroux, Luigi Liquori, Alberto Ciaffaglione [4].
- Oasis, INRIA Sophia: Bernard Serpette.

2 Présentation générale

Cette proposition d'ARC vise l'étude formelle des problèmes habituellement rencontrés par les concepteurs de compilateurs optimisants, souvent bien plus complexes que les compilateurs

1. `Catherine.Dubois@iie.cnam.fr`
2. `Sandrine.Blazy@iie.cnam.fr`
3. `boite@iie.cnam.fr`
4. `Alberto.Ciaffaglione@dimi.uniud.it`

“de référence” étudiés par les chercheurs spécialisés dans l'étude de la sémantique des langages de programmation: optimisation de l'utilisation de registres, de la gestion de la mémoire, de la gestion de la pile, et de l'utilisation des caractéristiques physiques des micro-processeurs. À ce titre, nous espérons réaliser des avancées significatives par rapport au résultats séminaux sur la démonstration de propriétés de langages, qui abordaient les problèmes de la vérification de compilateurs pour des langages réduits à l'épure, avec des machines cibles correspondant à des machines abstraites fournissant des fonctionnalités minimales.

2.1 Motivations

La certification de compilateurs est une problématique où se rejoignent d'un côté une demande clairement exprimée de la part des utilisateurs industriels de méthodes formelles, et de l'autre des travaux de recherche sur la preuve sur machine de propriétés sémantiques.

Les méthodes formelles (preuves de programmes, model checking, etc) sont de plus en plus appliquées au logiciel embarqué critique. Parmi les domaines d'activités où les méthodes formelles font une percée remarquable, citons les transports (avionique, automobile) et la banque (cartes de paiement). Une des raisons de cette percée est que les standards de certification de logiciels exigent, pour leurs niveaux les plus élevés, l'utilisation de spécifications formelles sur machine, voire de preuves de correction sur machine. C'est le cas pour les certifications Critères Communs de niveaux EAL6 et EAL7, ou les certifications CMM de niveau 5.

Nos discussions avec des industriels qui mettent en pratique les méthodes formelles (Dassault Aviation, Airbus Industries, EADS) font apparaître une difficulté: lorsqu'on fait une preuve sur machine ou du model checking d'un logiciel, c'est le code source (généralement écrit en C) qui est certifié; mais le code qui tourne effectivement dans le système embarqué est le code machine produit par la compilation de ce code source. Un compilateur incorrect peut parfaitement introduire des erreurs dans un code source soigneusement prouvé. Les bugs dans les compilateurs ne sont pas rares, surtout aux niveaux d'optimisation supérieurs, et surtout dans les compilateurs spécialisés pour les systèmes embarqués, qui sont moins bien testés que des compilateurs grand public.

En d'autres termes, lorsqu'on vise des niveaux de certification élevés, le compilateur C apparaît comme le maillon faible entre d'un côté le code source sur lequel on a appliqué des méthodes formelles, et de l'autre le microprocesseur qui bien souvent a été lui aussi certifié formellement.

Pour répondre à ce problème, la pratique courante dans l'industrie est de mener des revues manuelles du code assembleur produit par le compilateur, en le corrélant avec le code source. De telles revues sont très coûteuses en temps, assez peu fiables (une erreur humaine est vite arrivée, surtout lorsqu'il s'agit de relire un million de lignes de code assembleur), doivent être refaites à chaque changement du programme source ou du compilateur et interdisent l'utilisation de compilateurs optimisants.

2.2 Solution du compilateur certifié

L'alternative à cet état de fait peu satisfaisant est d'utiliser des compilateurs qui soient eux-même certifiés au même niveau de certification que le logiciel. En particulier, aux niveaux de certification les plus élevés, un compilateur certifié doit être accompagné d'une preuve formelle que le code machine produit est sémantiquement équivalent au code source, afin que toutes les propriétés formellement établies sur le source restent vraies pour le code machine compilé (cette notion de compilateur certifié est proche, mais distincte des compilateurs certifiants utilisés dans le contexte du code auto-certifié (*proof-carrying code*). Un compilateur certifié possède une preuve de correction valide pour tous les programmes d'entrée. Un compilateur certifiant produit pour chaque programme d'entrée non seulement du code machine, mais aussi une preuve de correction de ce code machine. Le compilateur lui-même n'a pas besoin d'être certifié).

C'est ici que les besoins de l'industrie rejoignent des travaux de nature plus universitaire menés depuis près de 20 ans sur la formalisation sur machine des sémantiques des langages de programmation, et la preuve sur machine de propriétés de ces langages (comme par exemple la sûreté du typage statique) et de la correction d'analyses et de transformations de programmes. Tous ces travaux constituent autant de briques de base nécessaires à la certification d'un compilateur. Cependant, ces travaux universitaires n'ont pas encore débouché sur la certification d'un compilateur complet, réaliste, et utilisable dans le contexte des logiciels embarqués. Typiquement, ces travaux s'intéressent à des langages sources minimalistes, des langages cibles et des schémas de compilation qui excluent toute optimisation.

La technologie de la preuve sur machine et de son utilisation pour raisonner sur les langages de programmation ont considérablement progressé dans les 10 dernières années, à un point tel qu'il nous semble que le moment est venu de tenter l'expérience de la certification en Coq d'un compilateur réaliste. Par "réaliste", nous entendons un compilateur qui compile un langage source utilisable en pratique (typiquement, le sous-ensemble de C utilisé pour programmer des systèmes embarqués); qui produit du code machine pour un microprocesseur RISC du monde réel; et qui effectue quelques optimisations parmi les plus rentables.

3 Description des activités scientifiques envisagées

3.1 Architecture d'un compilateur et de sa certification

L'architecture générale d'un compilateur est bien connue et n'a guère changé en 30 ans: après une analyse syntaxique et éventuellement une phase de vérification des types, une série de transformations de programmes fait progressivement passer du langage source au langage machine, via un ou plusieurs langages intermédiaires.

Parmi les langages intermédiaires les plus utilisés se trouvent les langages à transfert de registres (RTL), également appelés "code 3 adresses": ce sont des langages impératifs dont les constructions de base correspondent directement à des instructions du processeur, mais opèrent sur des pseudo-registres en nombre arbitraire. D'autres formes plus contraintes de RTL, comme la forme SSA (*single static assignment*) ou les graphes de flot, sont aussi utilisées.

Certaines transformations font passer d'un langage à un autre langage de plus bas niveau: explicitation des v-tables (pour les langages à objets), sélection d'instruction (production du RTL), allocation de registres (RTL avec pseudo registres vers RTL avec registres réels), émission de code machine. D'autres transformations (les optimisations) restent dans le même langage intermédiaire, mais transforment le code pour le rendre plus efficace: propagation de constantes, invariants de boucles, déroulage de boucles, élimination des variables d'induction, ordonnancement d'instructions.

La plupart des transformations s'appuient sur le résultat d'analyses statiques du code, qui indiquent quelles transformations effectuer et dans quels cas elles vont préserver la sémantique du code. Ainsi, l'explicitation des v-tables s'appuie sur des analyses de flot de contrôle; la propagation des constantes et les optimisations de boucles s'appuient sur des analyses *dataflow*; l'allocation de registres, sur une analyse *dataflow* de durée de vie, complétée par le coloriage du graphe d'interférence résultant.

Que faut-il certifier dans tout cela? Essentiellement, les transformations de programme. L'analyseur syntaxique, généré à partir d'une grammaire formelle, est essentiellement correct par construction. La vérification statique de types n'a pas besoin d'être certifiée, sauf si le reste du compilateur exploite de manière cruciale les résultats du typage. En revanche, chacune des étapes de transformation de code doit être accompagnée d'une preuve d'équivalence sémantique entre son entrée et sa sortie. Cela implique bien sûr que chacun des langages considérés (source, intermédiaires, cible) possède une sémantique formelle. Lorsqu'une transformation exploite les résultats d'une analyse statique, cette dernière doit aussi être prouvée correcte: les résultats de l'analyse doivent constituer une approximation correcte du comportement dynamique du code.

La preuve sur machine des transformations et analyses statiques de code soulève naturellement les problèmes généraux de la preuve de programme: il faut représenter ces transformations et analyses sous forme d'objets manipulables dans le système de preuve. Nous pensons que la manière la plus directe d'arriver à ce résultat dans le contexte d'un compilateur est d'écrire directement les transformations et les analyses sous forme de fonctions Coq, sur lesquelles on peut directement raisonner en Coq, puis d'extraire le programme du compilateur en Caml à partir des fonctions Coq.

3.2 Choix des langages source, cible, et intermédiaires

Les différents langages (source, cible, intermédiaires) impliqués lors de la compilation peuvent être choisis parmi les catégories suivantes, du plus haut niveau au plus bas niveau:

Haut niveau: orienté-objet, fonctionnel. Ces langages se caractérisent par une sémantique très propre, mais une distance élevée à la machine, nécessitant davantage de transformations pendant la compilation.

Bas niveau: C ou sous-ensemble de C. Ces langages sont polyvalents: on les trouve aussi bien comme langage source (typiquement, pour le code système ou embarqué) que comme langage intermédiaire (compilation via C). Ils correspondent à une demande industrielle forte. Leur sémantique formelle est assez malpropre, mais reste proche de celle de la machine.

Machines virtuelles à pile. Les machines virtuelles peuvent apparaître aussi bien comme langage cible (compilation Java vers JVM interprétée, comme dans JavaCard), intermédiaire, ou source (compilateur JIT pour JVM ou .Net).

Langages intermédiaires RTL, SSA. Point de passage obligé pour produire du code machine optimisé, ces langages ont une sémantique très proche de celle du code machine.

Code machine pour un processeur RISC. On atteint enfin ici l'interface entre le logiciel et le microprocesseur. Nous retenons ici des processeurs RISC genre PowerPC plutôt que des CISC genre Pentium pour deux raisons: le PowerPC est très répandu dans le monde de l'embarqué, et la sémantique des RISC est beaucoup plus simple à décrire.

Comme le montre cette énumération, le choix est vaste. Dans le cadre de cette ARC, nous proposons de nous concentrer en priorité sur la combinaison suivante: un sous-ensemble de C comme langage source; RTL comme langage intermédiaire; le code machine PowerPC comme langage cible. Cette combinaison est celle qui répond le plus aux problèmes pratiques du logiciel embarqué, et se caractérise par un “fossé sémantique” pas trop large entre langages source et cible.

Ce choix n'exclut en rien – et même prépare – l'extension ultérieure des techniques que nous développerons à d'autres langages de plus haut niveau. En effet, les trois langages ci-dessus (C, RTL, machine), ainsi que les transformations entre ces langages, sont communs à de nombreuses chaînes de compilation, dont certaines sont chères aux participants de cette ARC.

3.3 Méthode de travail

L'objectif de cette ARC est de jalonner le parcours vers la réalisation d'un compilateur certifié complet et modérément optimisant. Dans cette optique, nous avons identifié les expériences et sous-tâches suivantes:

- Définir une sémantique formelle des différents langages source, cible, et intermédiaires. En particulier, une sémantique utilisable sur machine d'un sous-ensemble réaliste de C est un point de passage obligé. L'utilisation sur machine doit comporter un volet “utiisation dans les systèmes de preuve” et un volet “exécution symbolique” de façon à tester ces spécifications sur des programmes exemples.
- Écrire un compilateur modérément optimisant soit directement sous forme de fonctions Coq, soit dans le fragment Coq de Caml, c'est-à-dire dans un style purement fonctionnel et où toutes les récursions sont primitives ou clairement bien fondées. S'assurer que ces contraintes sur l'écriture du compilateur peuvent être conciliées avec une rapidité raisonnable de compilation.
- Prouver la correction de certaines passes de ce compilateur, choisies pour leur représentativité. Un exemple simple est la propagation de constantes. Un exemple plus élaboré est l'allocation de registres par coloriage de graphe.
- Développer des théories Coq d'usage général dans ce type de preuves: formalisation des treillis, de la résolution des équations *dataflow*, de certaines formes d'analyse par interprétation abstraite.
- Identifier et développer les outils Coq (tactiques, etc) adaptés aux types de preuves que nous rencontrerons.

4 Le *savoir-faire* des équipes concernés

Cristal. Les membres de l'équipe Cristal ont la maîtrise d'œuvre de l'implémentation Caml et possède une maitrise dans le domaine des compilateurs optimisants. Ils ont déjà l'expérience de l'utilisation de Coq pour démontrer la correction d'outils d'analyse.

CPR, CEDRIC-Cnam. Les membres de l'équipe CPR fournissent des compétences sur la formalisation en Coq de propriétés de langages de haut niveau.

Lemme. Les membres de l'équipe Lemme ont déjà effectué des démonstrations de propriétés pour un noyau de langage impératif et pour la machine virtuelle Java.

Miró. Les membres de l'équipe Miró ont travaillé sur la formalisation en Coq d'une grande variété de mini-langages visant des styles divers de programmation (fonctionnelle, impérative, à objets).

Oasis. Les membres de l'équipe Oasis maitrisent les techniques d'analyse statique (vérification de bytecode) et de compilation vers les machines virtuelles.

Mimosa. Les membres de l'équipe Mimosa maitrisent les techniques de compilation optimisée vers C ou vers les machines virtuelles (JVM).

5 Moyen financiers demandés

Nous proposons principalement de financer des stagiaires et des post-docs d'un an (environ 50 k€), organiser des réunions (environ deux réunion plénières par an) faire des présentations dans des conférences internationales (voyages 20 k€), et probablement organiser un petit colloque sur le sujet (5 k€).

6 Budgets actuels 2002 des projets impliqués (pour information)

- Cristal : 107 k€
- Lemme : 360 k€
- Miró : 49 k€
- Oasis : ??€
- Mimosa : 263 k€