Compilation de *confiance*

C. Dubois, Cedric-IIE

Dans la littérature : 3 classes de travaux :

- vérification de code objet
- synthèse de compilateur
- vérification de compilateur

Les travaux cités dans la suite ne constituent pas un état de l'art exhaustif et ne traitent que de l'aspect *vérification de la compilation ou du code compilé*

Vérification de code objet

- Exemple : le vérificateur M68020 de Yu

sémantique opérationnelle en NQTHM (interprète) (80 pages, 569 fonctions, pas complet)
Il prouve que le code implémente correctement un algorithme donné, spécifié avec NQTHM.

Application avec succès pour du code objet issu de C, Ada et AKCL Lisp (Sun-3)

Autre système construit par Yu pour le processeur Dec Alpha

Y. Yu. Automated Proofs of Object Code for a Widely Used Microprocessor. PhD thesis, The University of Texas at Austin, CS-TR-93-09, 1992.

- Samet : système pour prouver que des programmes Lisp sont correctement compilés et optimisés en Lap (assembleur du PDP-10)
$\Rightarrow$ traduction des codes source et objet du programme en une forme intermédiaire et démonstration de leur équivalence (séquences d'exécution identiques)
Equivalence : pas de preuve

Samet, H., Proving the correctness of heuristically optimized code
Communications of the ACM July 1978 Volume 21(7), 570-582

Approche similaire pour la vérification du système de protection primaire d'une centrale nucléaire : programmes écrits en PL/M-86 pour le processeur 8086 (application à 85 000 lignes de PL/M-86, découverte de problèmes dont un décrit comme une erreur significative du compilateur)

D. J. Pavey, L. A. Winsborrow: Demonstrating Equivalence of Source Code and PROM Contents. The Computer Journal 36(7): 654-667, 93

Idem : ABB Signals (Suède) pour la vérification du compilateur du langage Sternol (systèmes ferroviaires) : *reverse compiler* appliqué au code objet (2ème source) puis vérification par preuve de l'équivalence des 2 sources (programmes Sternol équivalents à la logique propositionnelle)

Modelling and Verifying Systems and Software in Propositional Logic. Gunnar Stalmarck and M. Saflund. IFAC SAFECOMP'90. London, UK. 1990.

Conclusion : Vérifier que le code objet satisfait la spécification de l'algorithme $\Rightarrow$ On vérifie le résultat de certaines compilations

Passage à des langages plus complexes ?

Pas ou peu d'optimisation

Compiler Compiler

Des spécifications formelles (ex. Palsberg pour un sous-ensemble Ada) mais pas de vérification mécanisée des générateurs (à ma connaissance)

Vérification de compilateur

Preuve de la préservation de la sémantique par compilation : diagramme qui commute

- 1ère contribution réaliste : Polak en 81 : correction partielle d'un compilateur non optimisant pour un sous-ensemble de Pascal, pour une machine à pile fictive proche de la B6700

Stanford Pascal Verifier

Sémantiques dénotationnelles pour source et cible + conditions de vérification (définies au moyen de pre et post conditions dans langage d'assertions (sans quantificateur)

W. Polak. Compiler Specification and Verification. LNCS 124. Springer–Verlag, 1981

Mais hypothèses non prouvées (d'après Young) + inconsistences dans les axiomes

• CLI (Computional Logic Inc.) : vérification d'un compilateur pour un sous-ensemble du lg Micro-Gypsy (entiers, booléens, caractères, tableaux unidimensionnels, if, loop, signaux, procédures récursives, expressions et affectations)
Cible : Piton pour processeur FM8502 (données typées + procédures récursives)

sémantiques opérationnelles (machines à états finis) exprimées en NQTHM preuve par induction sur les constructions du langage source

10 mois-hommes

A Mechanically Verified Language Implementation. In Journal of Automated Reasoning, Kluwer Academic Publishers, 5(4), 1989, pp. 461-492.

• dialecte Pascal $\longrightarrow$ assembleur micro-controleur (98)
Z, quelques preuves en PVS
formalisation portée en Prolog pour exécution
contexte industriel

David WJ Stringer-Calvert, Susan Stepney, Ian Wand: Using PVS to Prove aZ Refinement: A Case Study. FME 1997: 573-588.

- ProCoS (Provably Correct Systems) projet esprit

Preuve d'un compilateur pour un sous-ensemble séquentiel de Occam pour Transputer simplifié

Utilisation d'un interprète écrit dans le langage source pour définir la sémantique de la machine cible ⇒ **`preuve du compilateur se réduit à montrer une relation de raffinement entre le code source et l'interprète pour le code généré`**

Utilisation du Boyer-Moore prévue initialement : succès ?????

Bowen - Hoare - Langmaach - Olderog - Ravn
ProCoS I project (1989-1991)
ProCoS II project (1992-1995)
ProCoS-WG Working Group (1994-1997)

- Projet Verifix (état de l'art courant ??)

www.info.uni-karlsruhe.de/ verifix/

Karlsruhe, Kiel et Ulm

  - Karlsruhe : compilation correcte d'un langage C like vers code Dec Alpha : spécification du langage avec ASM (Gurevitch). Compilation vers langage intermédiaire puis compilation vers code DEC Alpha spécifiée par des règles de récriture + générateur de back end.

  - Kiel : développement d'un compilateur correct pour un sous-ensemble de common Lisp : traduction du code Lisp vers un langage impératif (similaire au précédent) puis traduction en code pour transputer.

- Uml : formalisation, structuration et automatisation des preuves de correction de compilateurs

  - Travaux fondés sur l'hypothèse suivante : *même schéma de compilation pour les constructions classiques, pour tous les langages*

  $\Rightarrow$ théories génériques pour la compilation certifiée de constructions variées (PVS) (différents styles de sémantiques + théorie des domaines, points fixes)

  - Vérification du compilateur ComLisp (5 passes successives (front end $\rightarrow$ génération de code pour Transputer) 18 000 interactions avec le prouveur, 36 hommes-mois

  - vérification formelle d'optimisations locales (schéma générique applicable à une large classe d'architectures)

W. Zimmermann, T. Gaul, On the construction of correct compiler back-ends : an ASM approach, Journal of Universal Compiler Science, 3(5), 97

- Verified Scheme : compilateur vérifié pour Scheme

L'implémentation comprend 3 éléments : Scheme $\longrightarrow$ bytecode intermédiaire, interpréteur pour le bytecode écrit dans un sous-ensemble de Scheme (PreScheme), PreScheme $\longrightarrow$ assembleur cible

Vérification formelle mais complètement formelle (se concentre sur les algos et les structures de données, pas sur l'implémentation)

Outils ??

Extension : pas un back end particulier mais une classe de back ends caractérisée par un ensemble de critères $\rightarrow$ tout back end qui remplit ces critères préserve la sémantique.

Joshua D. Guttman, John D. Ramsdell, Mitchell Wand: VLISP: A Verified Implementation of Scheme. Lisp and Symbolic Computation 8(1-2): 5-32 (1995)

• Verified Assembler : vérification d'un assembleur pour un sous-ensemble du langage Vista, pour le micro-processeur Viper avec le système HOL

programmes Vista $\longrightarrow$ langage intermédiaire non structuré (Visa) (4 mémoires infinies) $\longrightarrow$ assembleur Viper (1 seule mémoire de taille finie) $\longrightarrow$ code machine Viper (non implémenté)

Schémas de langages source et cible.
Théorèmes génériques : réutilisation possible pour des langages concrets au prix de quelques définitions supplémentaires et preuves de quelques hypothèses.

Paul Curzon. A verified Vista implementation. Technical Report 311, University of Cambridge, Computer Laboratory, September 1993.

- Boutin : noyau fonctionnel ML vers code CAM : en Coq
- Le compilateur d'un petit langage impératif d'Yves Bertot (Coq)
- Compilateur certifié de μJava en Isabelle (Strecker, Berghofer équipe Nipkow).

Martin Strecker, Formal Verification of a Java Compiler in Isabelle,Proc. Conference on Automated Deduction (CADE), 2002, LNCS 2392, Springer Verlag, 63–77

- Certification du compilateur v5 d'Esterel dans Coq. Delphine Kaplan-Terrasse

et certainement plein d'autres ....

(et aussi preuves de transformation de programmes par exemple
Bertot, Fraer et Blazy,
preuves de typage (algo W (Nipkow et Dubois), vérification de bytecode ....)