HAL
open science

Coq a dit : fromage tranché ne peut cacher ses trous *

Jean-Christophe Léchenet, Nikolai Kosmatov, Pascale Le Gall

To cite this version:

Jean-Christophe Léchenet, Nikolai Kosmatov, Pascale Le Gall. Coq a dit : fromage tranché ne peut cacher ses trous *. Vingt-septièmes Journées Francophones des Langages Applicatifs (JFLA 2016), Jan 2016, Saint-Malo, France. hal-01333605

HAL Id: hal-01333605

https://hal.science/hal-01333605v1

Submitted on 15 Sep 2016

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

Coq a dit : fromage tranché ne peut cacher ses trous *

Jean-Christophe Léchenet[1,2], Nikolai Kosmatov[1] & Pascale Le Gall[2]

1 : CEA, LIST, Laboratoire de Sûreté des Logiciels
PC 174, 91191 Gif-sur-Yvette Cedex
prenom.nom@cea.fr
2 : MICS, CentraleSupélec
Grande Voie des Vignes, 92295 Châtenay-Malabry
prenom.nom@centralesupelec.fr

Résumé

Le *slicing* est une technique permettant d'extraire, à partir d'un programme donné, un programme plus petit, appelé tranche ou *slice*, tel que le programme et sa *slice* aient un comportement identique vis-à-vis d'un critère donné (appelé critère de *slicing*). Vérifier des *slices* de programme plutôt que le programme original est alléchant, mais cela nécessite des bases théoriques pour à la fois garantir la correction des résultats et permettre un *slicing* efficace.

Ce travail apporte les justifications théoriques nécessaires pour vérifier une *slice* plutôt que le programme initial. Nous définissons une notion de *slicing* relaxé qui produit des *slices* de petite taille y compris en présence d'erreurs ou de non-terminaison, et prouvons un résultat de correction pour ce *slicing*, qui nous permet de faire la correspondance entre les différents cas de présence/absence d'erreurs dans la *slice* et dans le programme initial. Ces énoncés justifient l'application de la démarche de vérification sur les *slices* plutôt que sur les programmes originaux. Ce résumé étendu présente la formalisation de ce travail dans Coq.

Introduction

Le *slicing* est une technique introduite par Weiser [11] pour décomposer un programme en un programme plus simple, appelée *slice*, en analysant ses flots de contrôle et de données. Dans la définition classique, une *slice* est un programme exécutable extrait du programme initial qui préserve le comportement par rapport à un point d'intérêt appelé critère de *slicing*, donné généralement sous la forme d'une étiquette désignant un point du programme initial, ou d'un ensemble d'étiquettes. Lorsqu'on *slice* par rapport à un ensemble d'étiquettes L, on veut que, pour chaque instruction dans L, les variables de cette instruction prennent la même valeur dans la programme initial et dans la *slice*. Le *slicing* a de nombreuses applications, notamment la compréhension de programmes, la maintenance de logiciels, la compréhension de bogues [9, 10]. Ce travail est réalisé dans le cadre du *slicing* statique arrière intra-procédural.

Pouvoir raisonner sur la *slice* plutôt que sur le programme initial est un enjeu important lorsqu'on fait de la vérification. La propriété classique du *slicing*, qui pourrait justifier la vérification de la *slice* plutôt que celle du programme initial, stipule que les exécutions du programme et de sa *slice* s'accordent après chaque instruction préservée dans la *slice* sur les variables apparaissant dans cette instruction. Mais on peut facilement montrer que cette propriété ne tient pas en présence d'erreurs ou de non-terminaison, lorsque l'instruction échouant ou la boucle responsable de la non-terminaison n'est pas préservée. En effet, l'erreur ou la non-terminaison dans une instruction non-préservée peut empêcher l'exécution des instructions suivantes dans le programme initial, alors que dans la slice

*. Ce travail a été partiellement financé par le programme EU-FP7 (projet STANCE, bourse 317753).

ces instructions pourraient s'exécuter et provoquer une erreur. Une solution possible est d'ajouter des dépendances (comme, par ex., [5, 7, 8]) pour conserver la propriété classique de *slicing* au prix d'obtenir des slices beaucoup plus larges. Nous proposons au contraire de garder essentiellement les mêmes relations de dépendance (et donc les mêmes *slices*) et d'affaiblir la propriété de correction.

Ce résumé étendu s'appuie sur l'article [6]. Les contributions de ce travail comprennent : une notion de *slicing* relaxé pour un langage simple en présence d'erreurs et de non-terminaisons, une propriété de correction adaptée et la justification d'une méthode de vérification basée sur les *slices*. L'article [6] formalise et apporte une preuve papier des différents résultats. Ce résumé étendu s'intéresse davantage à la formalisation de ces résultats dans l'assistant de preuve Coq [2] disponible sur `http://kosmatov.perso.sfr.fr/nikolai/slicing/`.

Description de la formalisation Coq

Les principales étapes de la formalisation sont présentées ci-dessous.

Un langage simple, s'appuyant sur des expressions arithmétiques et booléennes classiques, est défini. Il comporte des affectations, des conditions et des boucles. Il comporte également des assertions utilisées pour modéliser et expliciter les erreurs. On suppose que des assertions sont ajoutées dans le programme (comme peut le faire automatiquement le greffon RTE de la plate-forme Frama-C [4] pour le langage C) pour empêcher les comportements indéfinis, aussi bien ceux qui auraient de toute façon provoqué un échec, comme les erreurs à l'exécution, que ceux qui auraient pu passer inaperçus. Chaque assertion est associée à une instruction qu'elle protège. Des assertions peuvent être aussi ajoutées par l'utilisateur. Ce langage simple, puisqu'il peut donner lieu à la fois à des erreurs et à des non-terminaisons, est suffisamment représentatif pour notre formalisation du *slicing*. La définition du langage se trouve dans le fichier prog.v.

La sémantique du programme est une sémantique à base de traces. On note $\mathcal{T}[\![p]\!]\sigma$ la trace du programme p à partir de l'état initial σ. Les erreurs peuvent être uniquement créées par des assertions d'après notre hypothèse ci-dessus, donc nous n'avons pas besoin de modéliser les erreurs lors de l'évaluation des expressions. La sémantique est définie dans le fichier semantics3.v.

Trois types de dépendances sont utilisés : de contrôle, de données et d'assertion. Les dépendances de contrôle relient une instruction aux instructions pouvant décider de son exécution. Dans notre langage simple, seuls les `if` et les `while` créent des dépendances de contrôle. Les dépendances de données relient une instruction aux affectations pouvant modifier la valeur d'une de ses variables. Ces deux types de dépendance sont classiques [1]. Ils sont définis respectivement dans control.v et data.v. Nous définissons en plus des dépendances d'assertion associant les assertions aux instructions qu'elles protègent, et qui permettent de s'assurer que l'hypothèse de protection faite plus haut reste valide dans les *slices*. Ces dépendances sont définies dans assert.v.

La *slice* d'un programme par rapport à un ensemble d'étiquettes L est définie comme l'ensemble des instructions de ce programme dont dépendent (directement ou indirectement) les instructions dans L, la relation de dépendance considérée étant l'union des trois dépendances définies plus haut. La *slice* est définie dans slice.v. Un *slicer* pour notre langage est extrait du développement Coq.

La projection $Proj_L(T)$ d'une trace T sur un ensemble d'étiquettes L restreint la trace T aux instructions dans L et aux variables apparaissant dans les instructions correspondantes [1]. Elle permet de comparer les traces du programme initial et de ses *slices*. Cette définition est contenue dans semantics3.v.

La propriété de correction de notre *slicing* est plus faible que la propriété classique en général. Nous récupérons la précision du résultat classique dans le cas particulier où le programme original termine sans erreur. Ce résultat, prouvé dans le fichier slice.v, est exprimé par le théorème 1.

Théorème 1 *Soit p un programme et q une slice relaxée de p. On note L l'ensemble des étiquettes*

de q. Pour tout état initial σ, et tout préfixe fini T de $\mathcal{T}[\![p]\!]\sigma$, il existe un préfixe T' de $\mathcal{T}[\![q]\!]\sigma$ tel que : $Proj_L(T) = Proj_L(T')$. Si, de plus, p termine sans erreur sur σ, $Proj_L(\mathcal{T}[\![p]\!]\sigma) = Proj_L(\mathcal{T}[\![q]\!]\sigma)$.

La vérification sur les *slices* est justifiée par les théorèmes 2 et 3, qui découlent du théorème 1, eux aussi prouvés dans le fichier slice.v.

Théorème 2 *Si les assertions préservées dans la* slice *n'échouent jamais, elles n'échouent pas non plus dans le programme initial.*

Théorème 3 *Si une erreur est détectée dans l'exécution de la* slice *sur un état initial σ, alors trois cas sont possibles pour l'exécution du programme initial sur σ : la même erreur se produit, une erreur différente se produit dans une instruction non préservée ou l'exécution est infinie.*

Ces théorèmes relient précisément les cas de présence et absence d'erreurs dans la *slice* et dans le programme initial. Parmi les techniques de vérification s'appuyant sur le *slicing*, la méthode SANTE [3] permet de réduire la taille des programmes à vérifier de 51% en moyenne (allant même jusqu'à 97% pour certains exemples) et accélère la vérification de 43% en moyenne.

Conclusion et perspectives

Ce travail justifie l'utilisation du *slicing* pour vérifier des programmes qu'on ne peut supposer dépourvus d'erreurs ou de non-terminaisons [6]. Le présent résumé étendu présente une première formalisation en Coq, dont on peut extraire un *slicer* certifié. Ce développement réalisé pour un langage représentatif a requis 8 personnes-mois et représente environ 10.000 lignes de code Coq. Les perspectives de travail incluent une extension pour un langage réaliste, contenant notamment des appels de fonctions, et la certification d'une technique de vérification complète basée sur le *slicing*.

Références

[1] R. W. Barraclough et al. A trajectory-based strict semantics for program slicing. *Theor. Comp. Sci.*, 411(11–13) :1372–1386, 2010.

[2] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development.* Springer, 2004.

[3] O. Chebaro, N. Kosmatov, A. Giorgetti, and J. Julliand. Program slicing enhances a verification technique combining static and dynamic analysis. In *SAC 2012.*

[4] P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. Frama-C, a program analysis perspective. In *SEFM 2012.*

[5] M. Harman, D. Simpson, and S. Danicic. Slicing programs in the presence of errors. *Formal Aspects of Computing*, 8(4) :490–497, 1996.

[6] J.-C. Léchenet, N. Kosmatov, and P. Le Gall. Cut branches before looking for bugs : Sound verification on relaxed slices. In *FASE 2016.* A paraître.

[7] A. Podgurski and L. A. Clarke. A formal model of program dependences and its implications for software testing, debugging, and maintenance. *IEEE Trans. Software Eng.*, 16(9) :965–979, 1990.

[8] V. P. Ranganath, T. Amtoft, A. Banerjee, J. Hatcliff, and M. B. Dwyer. A new foundation for control dependence and slicing for modern program structures. *ACM Trans. Program. Lang. Syst.*, 29(5), 2007.

[9] J. Silva. A vocabulary of program slicing-based techniques. *ACM Comput. Surv.*, 44(3) :12, 2012.

[10] F. Tip. A survey of program slicing techniques. *J. Prog. Lang.*, 3(3), 1995.

[11] M. Weiser. Program slicing. In *ICSE 1981.*