Génération aléatoire de programmes guidée par la vivacité

Gergö Barany, Gabriel Scherer

Génération aléatoire de programmes guidée par la vivacité

Gergö Barany[1] et Gabriel Scherer[2]

[1] Inria Paris
gergo.barany@inria.fr
[2] Inria Saclay
gabriel.scherer@gmail.com

Résumé

Les programmes générés aléatoirement sont un bon moyen de tester des compilateurs et des outils d'analyse de logiciel. Des centaines de bogues ont été trouvés dans des compilateurs C très utilisés (GCC, Clang) par des tests aléatoires. Pourtant, les générateurs existants peuvent générer beaucoup de code mort (dont les résultats ne sont jamais utilisés). Compiler un tel programme laisse relativement peu de possibilités d'exercer les optimisations complexes du compilateur.

Pour résoudre ce problème, nous proposons la génération aléatoire de programmes guidée par la vivacité. Dans cette approche, le programme aléatoire est construit *bottom-up*, en combinaison avec une analyse de flot de données structurelle pour assurer que le système ne génère jamais de code mort.

L'algorithme est implémenté dans un greffon pour l'outil Frama-C. Nous l'évaluons en comparaison avec Csmith, le générateur aléatoire standard pour le langage C. Les programmes générés par notre outil compilent vers une plus grande quantité de code machine, avec une plus grande variété d'instructions.

Ce papier est une version courte d'un article présenté à LOPSTR 2017.

1 Motivation

Les compilateurs pour les langages de programmation modernes sont compliqués et difficiles à comprendre. Malgré les avances en vérification de compilateurs [Ler09, TMK+16], la plupart des compilateurs réalistes ne sont pas vérifiés formellement. Il faut des tests pour acquérir une confiance en leur correction partielle. Une approche populaire est le test aléatoire, fait avec des fichiers d'entrée produits par un générateur aléatoire de programmes. L'outil le plus connu dans ce domaine est Csmith, un générateur aléatoire de programmes C. Csmith est très puissant : il a trouvé des centaines de bogues dans des compilateurs très utilisés comme GCC et Clang, et même quelques bogues dans des parties non vérifiées du compilateur vérifié CompCert [YCER11].

Le travail présenté ici provient d'un projet de test aléatoire des optimisations effectuées par des compilateurs C. Nous avions besoin d'un générateur aléatoire produisant de grandes fonctions effectuant des calculs compliqués, sans appels de fonctions, pour pouvoir analyser le code machine optimisé généré par des compilateurs différents. Csmith peut être configuré pour cet usage, mais nous avons découvert que le compilateur enlève presque tous les calculs présents dans le code source généré par Csmith. Même des boucles imbriquées entières disparaissent souvent dans le code machine compilé à partir du code source généré par Csmith. Ceci nous empêche de faire des analyses intéressantes sur le code machine final.

Le problème vient du fait que Csmith génère des programmes qui ne contiennent presque que du *code mort* : des calculs dont les résultats ne sont jamais utilisés. Les compilateurs enlèvent tout le code mort, laissant très peu de possibilités d'appliquer nos analyses.

Cet article décrit `ldrgen`, un nouveau générateur de code C guidé par une analyse de vivacité pour éviter de générer du code mort.

2 Analyse de la vivacité

Une variable est *vivante* à une position donnée d'un programme si sa valeur à cette position peut être utilisée plus tard. Sinon, elle est *morte*. De la même façon, une affectation $v = e$ est vivante si elle définit une variable vivante, et morte sinon. Par exemple, dans un morceau de code comme `x = y + z; return y`, la variable `x` est morte juste après son affectation car sa valeur n'est jamais utilisée. L'affectation elle-même est donc morte aussi, et elle peut être enlevée du programme sans en changer le sens.

Notre objectif est de ne générer aucun code mort, et en particulier, de ne générer que des programmes dans lesquels toutes les affectations sont vivantes.

2.1 Analyse itérative

L'analyse de la vivacité est une analyse classique de flot de données [NNH99]. Elle parcourt le graphe de flot de contrôle du programme en arrière et itère jusqu'à trouver un point fixe. Chaque instruction S est liée à deux ensembles de variables vivantes : l'ensemble $S^\bullet$ de variables vivantes juste avant S, et l'ensemble S° de celles vivantes juste après S. Ces ensembles sont liés par des fonctions f_S spécifiques à l'instruction S qui décrivent la propagation de l'information. Pour une affectation $v = e$, à la variable v, de la valeur d'une expression e contenant l'ensemble de variables $FV(e)$, la propagation est faite par l'équation suivante :

$$S^\bullet = f_{v=e} = (S^\circ \setminus \{v\}) \cup FV(e)$$

Pour une instruction de branchement avec une condition c (comme $S =$ `if` (c) $\ldots$), l'analyse unit l'information de tous les successeurs dans le graphe de flot de contrôle et rajoute les variables de la condition :

$$S^\bullet = \bigcup_{S_i \in succ(S)} S_i^\bullet \cup FV(c)$$

Les boucles génèrent des systèmes d'équations récursifs. Ces systèmes sont traditionnellement résolus par itération jusqu'à obtenir un point fixe minimal comme solution.

2.2 Analyse structurelle de programmes sans code mort

L'analyse de flot de données sur le graphe de flot de contrôle est pratique pour identifier le code mort dans un programme donné. Cependant, notre objectif n'est pas d'analyser des programmes donnés mais d'utiliser l'analyse pour guider la génération de code afin d'éviter de générer du code mort. Nous voudrions éviter de construire un graphe de flot de contrôle et de calculer les points fixes pour les boucles par itération.

Nous avons donc décidé d'essayer une approche structurelle basée sur la structure de l'arbre de syntaxe abstraite. Les approches structurelles ne sont appropriées qu'aux programmes structurés (sans `goto` etc.), mais cela suffit pour notre générateur.

L'analyse est présentée dans la figure 1 comme un système de règles d'inférence. Les informations inférées sont des triplets $\langle S^\bullet \rangle \ S \ \langle S^\circ \rangle$ d'une instruction S et deux ensembles de variables vivantes avant et après l'instruction $(S^\bullet, S^\circ)$. Les règles contiennent deux sortes d'hypothèses : L'une sont les équations de propagation de l'information de l'analyse de flot de données. En particulier, la règle ASSIGN pour les affectations de la forme $v = e$ a comme hypothèse l'équation $S^\bullet = (S^\circ \setminus \{v\}) \cup FV(e)$, comme dans le cas de l'analyse de flot de données itérative.

La deuxième sorte d'hypothèses sont les conditions pour assurer que les programmes qui contiennent du code mort ne sont pas acceptés par le système. La condition la plus importante

$$\textsc{Return}\ \frac{}{\langle \{v\} \rangle\ \texttt{return}\ v\ \langle \emptyset \rangle} \qquad \textsc{Skip}\ \frac{}{\langle L \rangle\ \{\}\ \langle L \rangle}$$

$$\textsc{Assign}\ \frac{v \in S^\circ \qquad S^\bullet = (S^\circ \setminus \{v\}) \cup FV(e)}{\langle S^\bullet \rangle\ v = e\ \langle S^\circ \rangle}$$

$$\textsc{Sequence}\ \frac{\langle S_1^\bullet \rangle\ S_1\ \langle S_2^\bullet \rangle \qquad \langle S_2^\bullet \rangle\ S_2\ \langle S_2^\circ \rangle \qquad S_2^\bullet \neq \emptyset}{\langle S_1^\bullet \rangle\ S_1\ ;\ S_2\ \langle S_2^\circ \rangle}$$

$$\textsc{If}\ \frac{\langle S_1^\bullet \rangle\ S_1\ \langle S^\circ \rangle \qquad \langle S_2^\bullet \rangle\ S_2\ \langle S^\circ \rangle \qquad S^\bullet = S_1^\bullet \cup S_2^\bullet \cup FV(c) \qquad S_1 \neq \{\} \vee S_2 \neq \{\}}{\langle S^\bullet \rangle\ \texttt{if}\ (c)\ S_1\ \texttt{else}\ S_2\ \langle S^\circ \rangle}$$

$$\textsc{While}\ \frac{\langle B^\bullet \rangle\ B\ \langle B^\circ \rangle \qquad B^\circ = S^\bullet\ \text{(minimal)} \qquad S^\bullet = S^\circ \cup B^\bullet \cup FV(c) \qquad S^\circ \neq \emptyset}{\langle S^\bullet \rangle\ \texttt{while}\ (c)\ B\ \langle S^\circ \rangle}$$

FIGURE 1 – Système de règles d'inférence pour la reconnaissance de programmes sans code mort

est celle de la règle ASSIGN : Une affectation d'une variable v ne peut être acceptée que si v est vivante après l'affectation. De la même façon, la condition $S_1 \neq \{\} \vee S_2 \neq \{\}$ pour IF assure que les instructions de branchement inutiles comme `if (x) {} else {}` ne sont pas acceptées.

Le traitement des boucles est plus compliqué à cause de la dépendance cyclique entre l'ensemble $B^\bullet$ des variables vivantes au début du corps B de la boucle, et l'ensemble B° des variables vivantes à la fin du corps de la boucle. La condition de minimalité exprime que la solution désirée est un point fixe minimal du système d'équations. Un point fixe unique minimal existe toujours [NNH99].

Un programme S ne contient pas de code mort si le triplet $\langle S^\bullet \rangle\ S\ \langle \emptyset \rangle$ peut être dérivé dans le système de règles d'inférence. Il s'agit ici d'une approximation syntaxique : Les optimisations sémantiques du compilateur peuvent quand-même trouver des simplifications qui peuvent rendre inutile certaines parties du code.

3 Génération de code non mort

Les règles d'inférence peuvent être interprétées comme un générateur exécutable aléatoire (ou exhaustif) de programmes sans code mort. Comme l'analyse traditionnelle de vivacité, la génération se fait en arrière, c'est-à-dire dans la direction opposée au flot de contrôle.

Le générateur commence par générer une variable v et une instruction `return` v. L'ensemble de variables vivantes avant cette instruction est $L = \{v\}$. Le générateur applique des fonctions pour générer de nouvelles instructions aléatoires étant donné un ensemble de variables vivantes. Les nouvelles instructions S sont préfixées au programme, et l'ensemble L est mis à jour selon la fonction correspondante f_S. Cet ensemble guide le générateur : en particulier, pour générer une affectation d'une variable v, L doit contenir v à ce point dans le programme. L'itération se termine après un nombre prédéfini d'instructions, ou si jamais l'ensemble L devient vide. La figure 2 contient le pseudo-code du générateur dans un langage fonctionnel.

Lors de la génération des boucles, nous voudrions éviter la construction d'un graphe de

```
let random_statements L code =
    if L = ∅ then (code, L)
    else
        let (S, L') = random_statement L in
        random_statements L' (S :: code)

let random_statement L =
    let statement_generator = random_select [assignment; branch; loop] in
    statement_generator L

let assignment L =
    let v = random_select L in
    let e = random_expression () in
    ("v = e", (L \ {v}) ∪ FV(e))

let branch L =
    let (t, L1) = random_statements L [] in
    let (f, L2) = random_statements L [] in
    let c = random_expression () in
    ("if (c) t else f", L1 ∪ L2 ∪ FV(c))

let loop L =
    (* Générer un point fixe pour la boucle, puis la boucle elle-même. *)
    let c = random_expression () in
    let B' = random_variable_set () in
    let (code, L') = random_statements (L ∪ B' ∪ FV(c)) [] in
    let V = {b ∈ B' | b ∉ L' ∨ b n'a pas d'occurrences dans code} in
    if V = ∅ then
        ("while (c) code", L' ∪ L)
    else
        let e = random_expression_on_variables V in
        let v = random_select L' in
        let code' = "v = e" :: code in
        ("while (c) code'", (L' \ {v}) ∪ V ∪ L)

(* Commencer la génération à la fin du programme. *)
let v = random_variable ()
let (code, L) = random_statements {v} ["return v"]
```

FIGURE 2 – Pseudo-code du générateur aléatoire guidé par la vivacité.

flot de contrôle et l'itération jusqu'à un point fixe de la boucle. En effet, nous ne pouvons pas facilement faire une analyse itérative d'un corps de boucle qui n'existe pas encore.

Pour éviter ce problème cyclique, notre générateur choisit d'abord un ensemble $B^\bullet$ de variables qui doivent devenir vivantes au début du corps de la boucle, puis génère le corps lui-même. Finalement, il peut générer des instructions supplémentaires pour assurer la validité du choix de variables.

Plus précisément, nous cherchons à générer une liste B d'instructions et un ensemble $B^\bullet$ tels que $B^\bullet = f_B(B^\circ)$ et $B^\circ = S^\circ \cup B^\bullet \cup FV(c)$, étant donnés S° et une expression c pour la condition de la boucle. Nous générons un ensemble B' de nouvelles variables pour représenter les variables intéressantes pour le cas des boucles : les variables utilisées dans la boucle avec une valeur qui peut venir d'une itération précédente de la même boucle. Dit autrement, ces variables peuvent être affectées par le corps de la boucle, mais elles doivent aussi être utilisées dans le corps de la boucle.

Après avoir choisi l'ensemble B', nous pouvons générer une liste d'instructions sous l'hypothèse que les variables dans l'ensemble $S^\circ \cup B' \cup FV(c)$ sont vivantes à sa fin. Le générateur rend cette liste d'instructions (appelée *code* en figure 2) et l'ensemble L' de variables vivantes à son début. Pour que *code* puisse servir comme corps de boucle, il suffit d'assurer que notre choix de B' satisfait la condition ci-dessus : chaque variable $b \in B'$ doit être utilisée, et elle doit être vivante au début de la boucle. Il suffit de préfixer *code* avec une instruction qui utilise toutes les variables b ne satisfaisant pas encore cette condition pour obtenir une liste finale B et un ensemble $B^\bullet$ qui satisfont $B^\bullet = f_B(B^\circ)$, $B^\circ = S^\circ \cup B' \cup FV(c)$, et $B' \subseteq B^\bullet$.

Par exemple, imaginons vouloir générer une boucle `while` étant donné un ensemble $S^\circ = \{\mathtt{x}\}$ de variables vivantes après la boucle. Nous générons une condition aléatoire $c = \mathtt{y} < 10$ et un ensemble aléatoire $B' = \{\mathtt{z}\}$ de nouvelles variables à utiliser dans la boucle. Cela nous donne l'instance suivante de la règle While :

$$\frac{\langle B^\bullet \rangle \;\; B \;\; \langle B^\circ \rangle \qquad B^\circ = S^\bullet \qquad S^\bullet = \{\mathtt{x}\} \cup B^\bullet \cup \{\mathtt{y}\} \qquad B' = \{\mathtt{z}\} \subseteq B^\bullet}{\langle S^\bullet \rangle \;\; \mathtt{while}\ (\mathtt{y} < 10) \;\; B \;\; \langle \{\mathtt{x}\} \rangle}$$

Il reste à générer le code B pour le corps de la boucle sous l'hypothèse $B^\circ = \{\mathtt{x}, \mathtt{y}, \mathtt{z}\}$. Si le générateur renvoie le code `x = z + x; y = y + 1;`, la variable `z` est vivante à son début, et toutes les conditions sont satisfaites. Nous avons donc construit une dérivation du triplet

$$\langle \{\mathtt{x}, \mathtt{y}, \mathtt{z}\} \rangle \;\; \mathtt{while}\ (\mathtt{y} < 10) \;\; \{\mathtt{x} = \mathtt{z} + \mathtt{x}; \mathtt{y} = \mathtt{y} + 1;\} \;\; \langle \{\mathtt{x}\} \rangle \,.$$

En revanche, si `z` n'est pas vivante au début du code généré pour le corps de la boucle, il suffit de préfixer le code avec une affectation utilisant `z` en lecture.

4 Implémentation

Notre générateur `ldrgen` est implémenté dans un greffon pour Frama-C, qui est un outil extensible pour l'analyse et transformation de logiciels écrits en C [KKP+15]. `ldrgen` est un logiciel libre, disponible à https://github.com/gergo-/ldrgen. L'implémentation du générateur comprend environ 600 lignes d'OCaml, suivant la structure du pseudo-code en figure 2. Pour l'instant, l'outil génère un sous-ensemble du langage C avec des opérations arithmétiques et bit à bit sur tous les types de base, ainsi que des branchements avec `if`, des boucles `while`, et des boucles `for` d'une forme restreinte pour calculer une opération de réduction sur des tableaux de taille fixe. Il ne génère pas encore de `struct` ni d'arithmétique de pointeurs.

Une description plus complète se trouve dans la version originale de cet article [Bar17].

TABLE 1 – Comparaison du code généré par Csmith et `ldrgen` en 1000 appels de chacun.

	générateur	min	médiane	max	total
lignes de code	Csmith	25	368.5	2953	459021
	`ldrgen`	12	411.5	1003	389939
instructions	Csmith	1	15.0	1006	45606
	`ldrgen`	1	952.5	4420	1063503
opcodes uniques	Csmith	1	8	74	146
	`ldrgen`	1	95	124	204

5 Évaluation

Nous évaluons le code généré par `ldrgen` en comparaison avec Csmith. Pour ces expériences, nous avons configuré Csmith pour générer une seule fonction sans appels à d'autres fonctions. (Le mode standard de Csmith est de générer une application complète et de cacher la plupart des opérations arithmétiques dans des fonctions auxiliaires qui protègent des débordements.)

Nos résultats pour le code généré par 1000 appels de chaque outil sont résumés dans la table 1. La première partie de la table montre que nous avons choisi des options de configuration pour les deux outils pour générer des quantités comparables de code source. Ceci nous permet de faire une comparaison juste des programmes générés.

La deuxième partie de la table montre le nombre d'instructions machine émises par GCC (`-O3`, sur x86-64) pour le code source généré par chaque outil. La médiane de 15 instructions machine pour Csmith nous démontre qu'au moins la moitié des fonctions générés par Csmith ont une taille triviale. Ces fonctions très petites sont exactement ce que nous voulions éviter car elles ne sont pas intéressantes pour notre analyse de code machine. En moyenne, le code généré par Csmith compile vers environ une seule instruction machine par 10 lignes de code source. En revanche, avec `ldrgen` nous obtenons environ 2,5 instructions machine par ligne de code.

La dernière partie de la table concerne la variété d'instructions générées. Nous regardons les nombres d'instructions machine différentes par fonction générée et au total. Même la fonction la plus diverse générée par Csmith contient moins de variété que la fonction médiane générée par `ldrgen`. Au total, le code généré par `ldrgen` permet une couverture du jeu d'instructions 40 % plus élevée par rapport au code généré par Csmith. Par inspection des ensembles d'instructions uniques, nous avons trouvé que presque toute la différence vient des instructions vectorielles (SIMD) qui sont émises pour les boucles `for` sur des tableaux générées par `ldrgen`. Csmith sait générer des boucles similaires, mais leurs résultats ne sont presque jamais utilisés.

6 Conclusions

Nous avons présenté `ldrgen`, un générateur de code source C aléatoire. Le but de ce générateur est de ne jamais générer du code mort, afin d'obtenir des grandes quantités d'instructions machine variées générés par des compilateurs optimisants. `ldrgen` est guidé par une analyse structurelle de vivacité lors de la génération de code. Une nouvelle approche structurelle de l'analyse de flot de données, interprétée comme système d'inférence, nous permet de faire l'analyse sans devoir implémenter l'itération jusqu'à un point fixe. Par rapport au générateur de code Csmith, `ldrgen` génère une plus grande quantité de code machine plus varié.

Références

[Bar17] Gergö Barany. Liveness-driven random program generation. In *27th International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR 2017)*, 2017.

[KKP+15] Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-C : A software analysis perspective. *Formal Aspects of Comp.*, 27(3) :573–609, 2015.

[Ler09] Xavier Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7) :107–115, July 2009.

[NNH99] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[TMK+16] Yong Kiam Tan, Magnus O. Myreen, Ramana Kumar, Anthony Fox, Scott Owens, and Michael Norrish. A new verified compiler backend for CakeML. In *ICFP 2016*, 2016.

[YCER11] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in C compilers. In *PLDI '11*, pages 283–294. ACM, 2011.