Je suis pleinement conscient(e) que le plagiat de documents ou d'une partie de document constitue une fraude caractée.
RAYNAUD Paul, 8 juin 2018 et signature :

RAYNAUD Paul
.
Dirigé par : PÉRIN Michaël

Juin 2018

Compilateur C vers clauses de Horn

Résumé Vérifier de manière automatique si un programme réalise sa spécification nous permet de montrer quelque chose de plus fort que l'absence de certaines erreurs, nous sommes capables de montrer la correction du programme. Nous chercherons donc à prouver à partir d'un programme source si celui ci est capable de satisfaire les propriétés qui lui sont attachées ou non le tout de manière automatique. Nous nous baserons sur des principes déjà connus tel que la Logic de Hoare, et l'étude de graphe de flot de contrôle pour générer des clauses de Horn. Une fois ces formules logiques générées nous aurons recours à un SMT solver pour résoudre les-dites formules. Mon stage étant sur plus d'un mois (magistère) les résultats actuels se limitent à la mise en place de bonnes conditions pour travailler réellement sur le sujet.

Keywords CompCert · Program verification · Automatic Verification

1 L'intêret de vérifier un programme

1.1 Les risques liés à l'informatique

Les bugs sur des systèmes informatiques sont quelques choses de banalisés, mais l'informatique étant présente partout, et notamment dans des domaines critiques (santé, embarqué, ...), il est indispensable de prouver qu'un logiciel est capable de répondre aux contraintes décrites dans sa spécification.

Paul RAYNAUD
361 allée de Hector Berlioz 38400
Tel. : 06 15 54 38 90
E-mail: paul.raynaud66@hotmail.fr

1.2 Explication brève des choix du projet

L'objectif à long terme est de réaliser un outil capable de vérifier à la compilation si un programme satisfait des propriétés de sûreté telles que l'absence de division par zéro, l'absence de débordement de tableaux, ... Le language C est le plus utilisé dans les systèmes embarqués critiques, par conséquent nous travaillerons sur un vérificateur de programme C.

Les principes de vérification ont été décris pour la première fois en 1969 dans "An Axiomatic Basis for Computer Programming", par Charles A.R. Hoare inspiré des travaux de Floyd. La Logique de Hoare est un outil de vérification formelle d'un programme basé sur la sémantique du language dans lequel est écrit le programme. Il est naturellement plus simple d'utiliser ces principes sur des graphes de flot de contrôle pour générer les clauses de Horn. Un CFG est une représentation d'un programme permettant de visualiser les chemins qu'il peut atteindre. Les clauses de Horn sont des implications logiques quantifiées universellement dont les prémisses et la conclusion sont des prédicats de base. Ces clauses sont par ailleurs l'une des entrées standards utilisées par de nombreux SMT solvers. Un SMT-solver est un SAT-Solver, un logiciel prenant en entrée une formule logique comme les clauses de Horn et qui en sortie répond si cette formules est satisfaisable ou non. (ex Alt-Ergo)

La norme C est floue sur certains points et laisse place à l'interprétation, de ce fait la sémantique varie subtillement d'un compilateur C à un autre. D'autres parts il existe des erreurs de compilation appelées : miscompilations. Celles-ci génèrent des éxécutions incorrectes de programmes pourtant corrects. C'est pour cette raison que le compilateur CompCert a vu le jour, il est actuellemnt le seul compilateur C prouvé (en coq) , et c'est donc en nous appuyant sur ces deux arguments que nous baserons notre sémantique du C sur celle de CompCert.

2 Travaux Connexes

2.1 La vérification formelle et CompCert

Des outils capables de réaliser la vérification de programmes ont déjà été développés, comme SeaHorn. Cependant comme nous le verrons par la suite ils ne sont pas adaptés au "récent" (présenté au ERTS 2016 : Embedded Real Time Software and Systems, 8th European Congress, Jan 2016, Toulouse, France) compilateur CompCert. Avec sa sémantique unique, ce serait pour lui un nouvel atout d'être capable de vérifier automatiquement un programme. On pourrait ainsi avoir un programme prouvé sans qu'un compilateur à la sémantique "hasardeuse" vienne intégrer des erreurs lors de la compilation, sur un code dont la spécification est prouvée.

2.2 SeaHorn

Un outil comme SeaHorn repose sur les mêmes principes que ceux utilisés dans ce stage, à la différence que SeaHorn prend des programmes C/C++ et utilise le bitecode généré par Clang. Clang est un compilateur différent et par conséquent n'a pas la même sémantique que CompCert, le code généré diffère. C'est pour cette raison que nous ne pouvons pas utiliser SeaHorn.

La raison générale pour laquelle nous ne pouvons réutiliser les outils déjà existants réalisants de la vérification de programmes, est liée au fait que CompCert est un compilateur "récent" et que ces outils n'ont pas été developpés sur sa sémantique du C.

2.3 CompCert

Yang et al "Finding and Understanding Bugs in C Compilers"(2011) ont réalisé une étude visant a montré les miscompilation des différents compilateurs C, le résultat montré est que tous les compilateurs qu'ils ont testés ont généréf des codes incorrects pour certains tests (y compris CompCert). Rappelons que CompCert n'était pas terminé en 2011, mais il réduisait déjà des bogues par rapport aux autres compilateurs :

> "The striking thing about our CompCert results is that the middle- end bugs we found in all other compilers are absent. As of early 2011, the under-development version of CompCert is the only compiler we have tested for which Csmith cannot find wrong-code errors. This is not for lack of trying : we have devoted about six CPU-years to the task. The apparent unbreakability of CompCert supports a strong argument that developing compiler optimizations within a proof framework, where safety checks are explicit and machine-checked, has tangible benefits for compiler users".

CompCert est un projet démarré en 2008 visant à créer le premier compilateur C prouvé. C'est un projet à la structure complexe entièrement prouvé et majoritairement développé en coq. De ce code coq il est possible d'extraire du ocaml éxécutable. Et c'est sur ce ocaml que nous travaillerons.

3 Principe de la traduction

3.1 Résultat attendu

L'objectif global attendu de notre outil est explicité dans la Figure 1, nous détaillerons par la suite la traduction étape par étape.

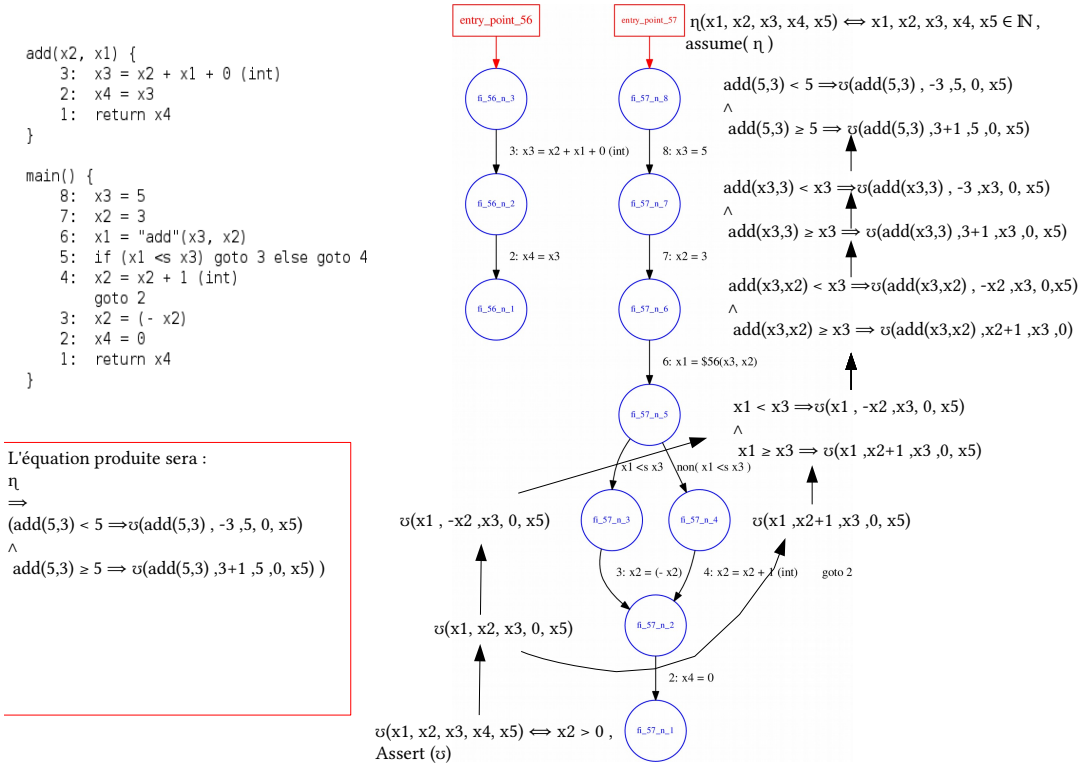

Figure 1 Graphe de Flot de Contrôle (CFG) avec le code c associé à côté et les équations logiques qui en découle, noter que toute la formule n'est pas noter a chaque étape.

3.2 D'un graphe de Flot de Contrôle à des formules logiques

Prenons un exemple plus simple dont le code C et sa traduction RTL sont définis dans la Figure 2 .

Nous allons premièrement rappeler les formules de la logique de Hoare :

Un triplet de Hoare est :

$$\{\rho\}programme\{\gamma\}$$

où ρ *représente la précondition, et* γ *la post* $-$ *condition du programme.*

Il existe 2 méthodes pour générer les formules logiques d'un programme :

- Strongest Postcondition, qui génère les formules en partant du début du programme vers la fin.

- Weakest Precondition, celle que nous utiliserons qui part de la fin du programme est qui remonte le programme.

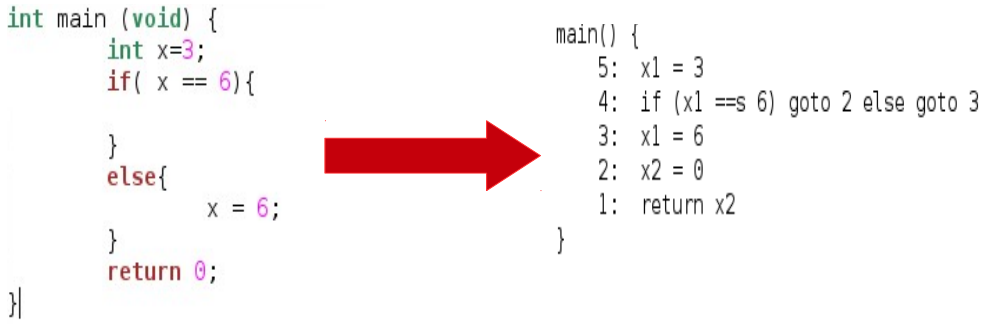

Figure 2 transformation du C au RTL

Nous allons maintenant expliciter les règles concernants la Weakest Precondition :

Un programme voulant être vérifié doit satisfaire :

$$\forall x, \rho \implies WP(programme, \gamma)$$

Une autre manière de l'écrire est la suivante :

$$WP : \frac{\rho \Longrightarrow WP(instr, \gamma)}{\{\rho\}\ instr\ \{\gamma\}}$$

γ *est la préconditionet*ρ *la postcondition.*

La séquence :

$$WP(S1; S2, \gamma) \ = \ WP(S1, WP(S2, \gamma)$$

L'affectation :

$$WP(x \ := \ e, \gamma) \ = \ \gamma[x \leftarrow e]$$

La condition

$$WP(\ if\ C\ then\ S1\ else\ S2,\ \gamma) \ = \ (C \Longrightarrow WP(S1, \gamma)) \wedge (\neg C \Longrightarrow WP(S2, \gamma))$$

Nous devons maintenant appliquer ces formules à notre graphe issu de la représentation RTL Figure 3.

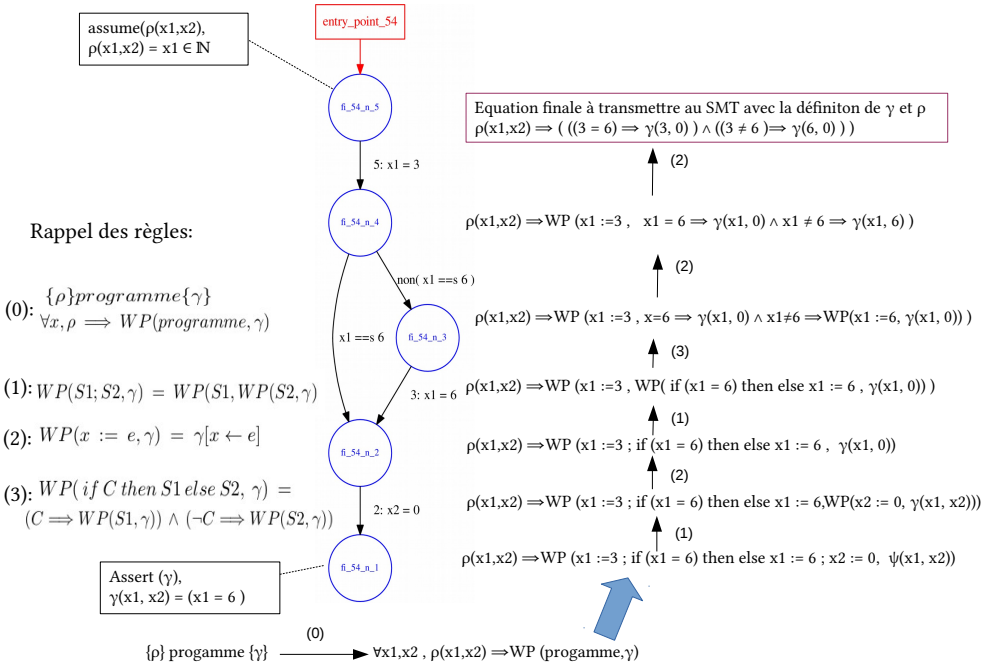

Figure 3 Construction avec les règles de WP

4 Contribution

4.1 principe théorique

Nous nous sommes intégrés dans CompCert au premier endroit où il créé un CFG, de manière à utiliser les principes évoqués précedemment, bien que la Logique de Hoare soit déjà définie, pour l'appliquer sur CompCert il faudra réécrire les règles pour les adapter aux instructions du language intermédiaire que nous exploitons : RTL (Figure 5).

4.2 Travail fourni

Premièrement il faut d'abord se rattacher à CompCert, un compilateur industriel, comprendre globalement l'architecture du logiciel et comment sont reliés entre eux les différents fichiers afin de finalement travailler dessus. Bien que nous ayons eu facilement un accès aux données nécessaires, cependant cela n'était pas suffisant pour les manipuler librement. En effet modifier quelque chose pour le tester revenait à recompiler CompCert, raison pour laquelle nous avons utilisé une librairie caml particulière (ppx_deriving.show) pour récupérer

la structure de données traité par CompCert. Le principe de la librairie est de générer automatiquement des fonctions capables d'afficher un type, qu'il soit un type somme, enregistrement ou autre. Pour cela nous avions besoin de modifier les fichiers où étaient définis ces types que nous voulions afficher, ainsi que les fichiers où l'on definissait les types de bases (Figure 4).

La compilation fût un peu compliquée (annexe 6.1) mais une fois la structure du code compilé par CompCert extraite, nous obtenons une liste de graphes de flot de contrôle, où chacun d'eux représente une fonction/procédure. (Figure 5)

En retriant les instructions de l'arbre, et en les modifiant nous avons réalisé une première nouvelle structure qui sied mieux à la représentation d'un CFG, une structure par triplet (Figure 6).

Une fois notre arbre de départ transformé en un CFG sous forme de triplet nous avons développé un printer au format .dot permettant d'afficher un graphe grâce au logiciel graphviz (graphe sur les Figures 1 et 3).

Maintenant que nous pouvons démarrer sur des bases plus solides, nous devrions developper la génération de clauses de Horn au format du SMT-solver que nous choisirons.

```
type test =
        | A of int
  [@@deriving show]
;;

type tree =
  | U of tree
  | B of tree * tree
  | L of int
        | T of test
  [@@deriving show]
;;
```

```
open Tree

let output_ocaml_decl_of_tree: string -> tree -> unit
=
  fun name tree ->
    (String.concat "\n" [ "open Tree" ; "let " ^ name
^ " =" ; show_tree tree ; ";;"])
    |> print_string
;;

let t = B(U(L 1), B (L 2, B( (L 3), (T(A 3)) ) )  ) ;;
let _ =
        output_ocaml_decl_of_tree "tree1" t ;;
```

```
sule:/local/raynaudp/TER/projet/CC2HC/ressource/generic_printer$ ocamlfind oca
mlc -c -package ppx_deriving.show tree.ml
sule:/local/raynaudp/TER/projet/CC2HC/ressource/generic_printer$ ocamlfind oca
mlc -o exec -linkpkg -package ppx_deriving.show tree.cmo main.ml
sule:/local/raynaudp/TER/projet/CC2HC/ressource/generic_printer$ ./exec
open Tree
let tree1 =
(Tree.B ((Tree.U (Tree.L 1)),
   (Tree.B ((Tree.L 2), (Tree.B ((Tree.L 3), (Tree.T (Tree.A 3))))))))
```

Figure 4 Exemple d'utilisation de ppx_deriving.show

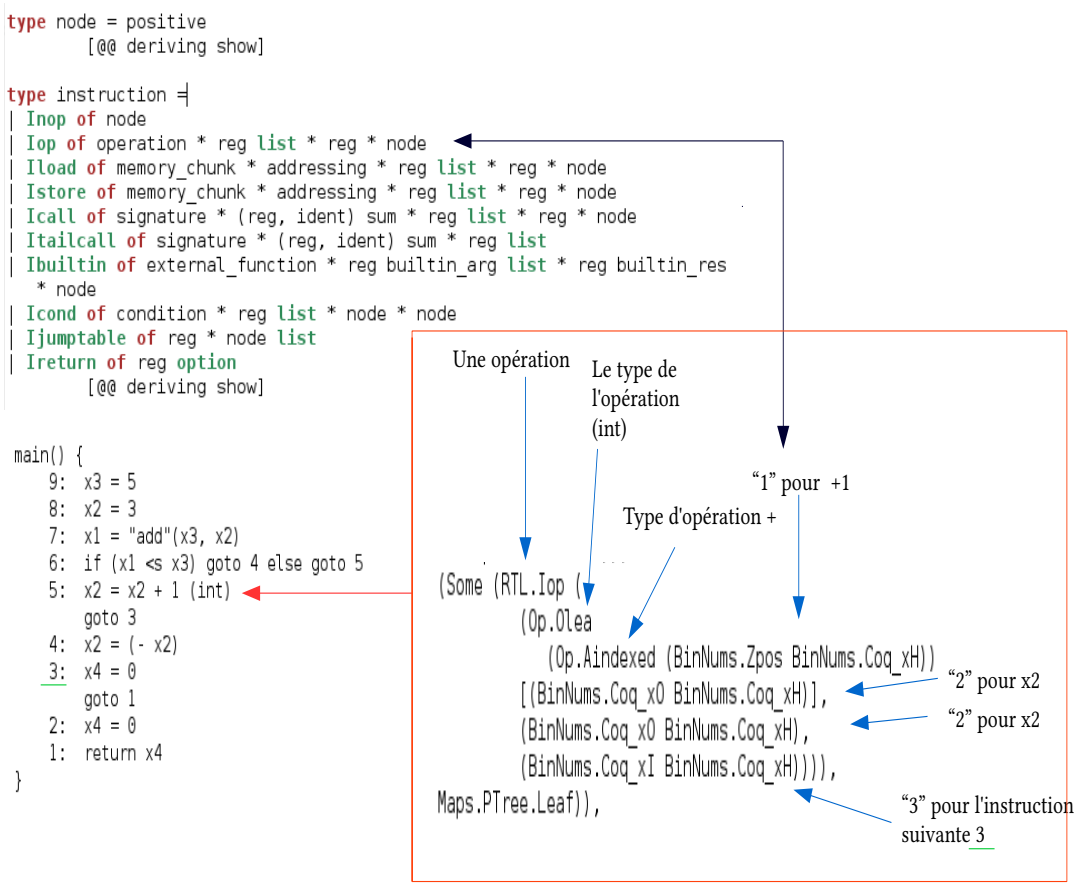

Figure 5 Description d'une opération

5 Extensions possibles et probables

5.1 À court terme

Le but de ce stage est de commencer ce compilateur du C vers des clauses de Horn en développant un premier outil capable de générer des clauses pour un sous-ensemble (plutôt simple) d'instructions de C. Mais ce n'est pas une finalité en soit.

```
type triplet ={ noeud_t : int; instr_t : RTL.instruction;
dir_t :int list}
  [@@ deriving show];;
```

Un triplet est concrètement une instruction, mais on rajoute comme information le numéro de son noeud et les noeud vers lesquels elle pointe

```
type 'a cfg = {ident : int; param : reg list; entry_point :
int ;instrs : 'a list}
  [@@ deriving show];;

type 'a cfg_complet = ('a cfg) list
  [@@ deriving show];;
```

-Un graphe de flow de contrôle est l'ensemble des instructions (triplet que nous avons défini précédemment) présent dans une fonction ou une instruction.

-Un graphe complet est l'ensemble des fonctions/instructions définies dans un programme, elles sont dans notre cas stocké sous forme de liste.

Figure 6 définition des triplets et cfg

5.2 Sur le moyen terme

Une seconde étape serait de compléter cet outil pour être capable de traiter tout le C, comme ce que réalise déjà les autres "verifieur" (SeaHorn, Boogie ...), car si on ne supporte pas toutes les instructions supportés par CompCert, l'outil sera inutilisable (ou peu utilisable).

5.3 Pour être synchronisé avec CompCert

Une fois que toutes les instruction du C acceptées par CompCert sont traitées au complet il faudra rester dans la même logique que CompCert et prouver ce que nous avons réalisé jusque là. D'après Mr. Périn ce serait le travail d'une thèse.

Si le projet arrivait à son terme, alors un programme compilé par CompCert (et notre outil) serait capable d'avoir des propriétés, et d'être sûr de les maintenir sans craindre de miscompilations.

6 Annexe

6.1 Une compilation compliqué

Cependant les fichiers caml (.ml) contenants les types (les structures) de base de la structure d'un programme et leurs interfaces (.mli), sont générés lors

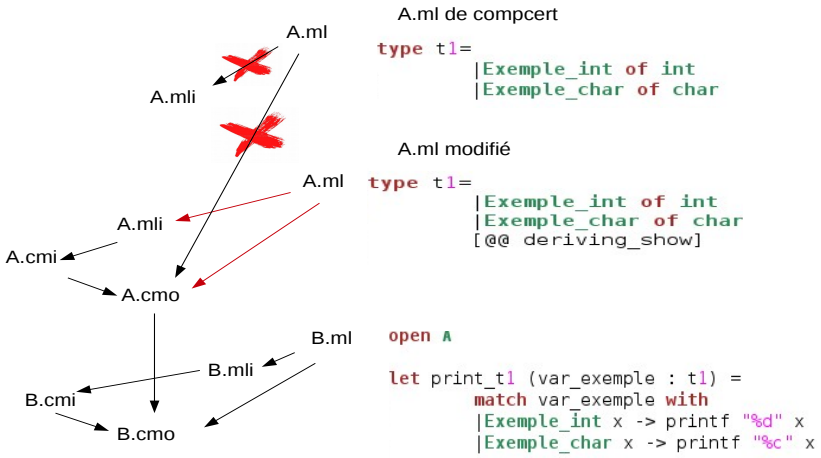

Figure 7 Exemple simple de la compilation

de la compilation (de CompCert), ce qui pose 2 problèmes : - l'effacement de nos fichiers à chaque recompilation de CompCert - la simple modification des .ml ne suffit pas, car l'interface n'est plus en accord avec le fichier caml. La création d'un nouvelle interface pour chaque fichier modifié est donc necessaire.

Nous avons donc décidé de sauvegarder nos fichiers modifiés (.ml) dans un répertoire particulier, les compiler de manière indépendante pour générer leurs interfaces et les fichiers intermédiaires (.mli, cmi). Car de la même manière que les .mli, le .cmi dépend des .mli et si nous utilisons les .mli générés par CompCert au même moment que les .ml, il y aura un conflit entre l'interface et le fichier source et il sera impossile de créer le .cmi qui est necessaire à la compilation. Une fois cela réalisé, nous les avons réinjecté au moment opportun dans la compilation de CompCert, pour qu'il créé lui même les .cmo et que l'on soit capable d'afficher la structure que nous voulons, en modifiant légèrement un printer déjà existant.(Figure7)

Références

[1] Yang Al ,"Finding and Understanding Bugs in C Compilers", pages 1-11, 2011
[2] C. A. R. HOARE ,"An Axiomatic Basis for Computer Programming", page 1-6, 1969