Alias : pointeurs espionnés en série

Tristan Le Gall, Jan Rochel, Florian Faissole, Julien Signoles, Denis Cousineau

Alias: pointeurs espionnés en série

Tristan Le Gall[2], Jan Rochel[2], Florian Faissole[1], Julien Signoles[2] et Denis Cousineau[1]

[1]Mitsubishi Electric R&D Centre Europe, Rennes, France
[2]Université Paris-Saclay, CEA, List, Palaiseau, France

Cet article présente un nouveau greffon de Frama-C, appelé Alias. Il s'agit d'une analyse de pointeurs légère destinée à être intégrée d'une manière transparente dans d'autres analyseurs. Elle doit donc être à la fois correcte, efficace et ne requérir aucun paramétrage de la part de l'utilisateur. À cette fin, nous avons adapté l'algorithme de Steensgaard pour développer une analyse sensible au contexte à l'aide de résumés de fonction, sensible aux champs de structure, mais insensible aux tableaux et, plus généralement, à l'arithmétique de pointeurs. Nos évaluations expérimentales sur un *benchmark* représentatif montrent qu'elle est bien plus rapide qu'Eva, l'analyse de valeurs de Frama-C préexistante qui fournit aussi une analyse d'alias, tout en gardant une précision suffisante aux besoins.

1 Introduction

En programmation, les pointeurs sont très utiles. Ils dénotent en effet une adresse mémoire du programme, l'« adresse pointée », contenant une autre valeur, la « valeur pointée », ce qui permet d'effectuer simplement des opérations bas-niveau liées à la mémoire, en particulier lorsque cette dernière est allouée dynamiquement pendant l'exécution du programme. On peut les manipuler directement dans des langages permettant des opérations dites de bas-niveau, comme le langage C, ou bien indirectement, sous forme de références dans les langages fonctionnels comme OCaml, ou d'objets dans les langages orientés objets comme Java. L'expressivité ainsi permise a néanmoins un prix : elle rend la compréhension des programmes plus compliquée, que ce soit par un humain ou par un outil automatique. Cette difficulté provient en particulier du problème de l'*aliasing*, c'est-à-dire lorsque deux pointeurs dénotent la même adresse mémoire à un point de programme donné pendant son exécution. Ainsi, dans l'exemple suivant, les pointeurs `p` et `&x` sont aliasés à partir de la ligne 2.

```
int x = 0;
int *p = &x;      // p et &x sont maintenant en alias
x = 1;            // *p est egalement modifie
printf("%d", *p); // affiche 1
```

Détecter les pointeurs en alias et quelles adresses mémoires sont pointées par quels pointeurs est indécidable dans le cas général, alors même que ce type d'information est nécessaire à la correction de beaucoup de compilateurs ou d'analyseurs de code. Ainsi, dans la plateforme Frama-C dédiée à l'analyse de code C [BBB+21], une telle analyse de pointeurs serait utile pour des tâches diverses, comme optimiser du code généré par transformation de programmes ou des obligations de preuve émises par un outil de vérification déductive, ou encore analyser des propriétés sur des programmes concurrents.

Rapellons que Frama-C est composée d'un noyau (qui fournit un parseur du langage C et divers moyens de parcourir et de transformer l'AST obtenu) et de multiples greffons qui sont spécialisés dans un type d'analyse ou de transformation. Frama-C dispose de son langage de spécification formelle : ACSL [BCF+]. C'est une plateforme *open-source* et qui permet à tout un chacun de développer de nouveaux greffons, chaque greffon pouvant utiliser les fonctions du noyau ainsi que celles des autres greffons (via leur API).

Frama-C dispose donc déjà d'une analyse d'alias. En effet, son greffon Eva [BBY17] fournit une analyse de valeurs par interprétation abstraite [Cou22, CC77] qui calcule une sur-approximation correcte des valeurs possibles pour chaque expression C en tout point du programme : les adresses mémoires et les pointeurs étant des expressions C particulières, cette analyse est donc, entre autres, une analyse de pointeurs. Elle est très puissante, pouvant notamment passer à l'échelle sur des programmes de grande taille [Our15] tout en donnant des résultats précis. Néanmoins, pour y parvenir, l'utilisateur doit paramétrer finement l'analyseur, ce qui requiert une expertise certaine, afin de régler au mieux les nombreux compromis possibles entre précision d'un côté et rapidité et consommation mémoire de l'autre. Ces paramétrages et cette expertise font que Eva *n'*est *pas* l'outil adéquat lorsqu'on souhaite *uniquement* s'en servir, dans une autre analyse, pour obtenir les alias et/ou les adresses pointées par un pointeur. En effet, dans un tel contexte, on souhaite en général d'une part une analyse transparente pour l'utilisateur, ne requérant en particulier aucun paramétrage spécifique, et d'autre part, une analyse particulièrement rapide, puisqu'elle est destinée à être englobée dans une analyse plus importante, souvent elle-même coûteuse en temps d'exécution.

Cet article présente un nouveau greffon Frama-C, appelé Alias, qui répond à ce besoin en proposant une nouvelle analyse de pointeurs dite « légère », c'est-à-dire rapide et ne requérant aucun paramétrage utilisateur. Cette nouvelle analyse de pointeurs est principalement destinée à être utilisée par d'autres analyses *via* son API. Elle adapte l'algorithme de Steensgaard [Ste96]. Elle est sensible au contexte (*context sensitive*) pour être raisonnablement précise en présence d'appels de fonction, tout en demeurant efficace grâce à des résumés de fonction. Elle est aussi sensible aux champs des structures (*field sensitive*) pour rester précise lorsqu'on y accède. Elle est en revanche insensible aux tableaux (*array insensitive*) et, plus généralement, aux décalages calculés *via* de l'arithmétique de pointeurs, afin de beaucoup gagner en efficacité tout en ne subissant qu'une perte de précision limitée. L'efficacité en temps et en mémoire, ainsi que la précision, de ce nouveau greffon ont été évaluées sur un *benchmark* existant, déjà régulièrement utilisé pour évaluer Eva.

État de l'art L'analyse de pointeurs est un problème qui a fait l'objet de nombreux travaux de recherche (voir par exemple [SB15]), avec de grandes familles d'algorithmes bien identifiés. La première de ces familles est celle fondée sur l'algorithme d'Andersen [And94]. L'algorithme d'Andersen travaille avec des contraintes sur des sous-ensembles, tandis que Steensgaard [Ste96] utilise des contraintes d'égalité. Par conséquent, l'algorithme d'Andersen est généralement plus précis, mais aussi plus coûteux, que celui de Steensgaard. Quelques années plus tard, [DMM98] a proposé une analyse d'alias reposant sur le typage (statique) d'un programme, et qui n'est donc pas tout à fait adaptée à l'analyse du langage C, du fait des opérations de *casts* permettant des conversions dynamiques de types. Il existe également des analyses de pointeurs "*demand-driven*"[HT01] qui ne cherchent qu'à calculer une partie du graphe de pointeurs [SGSB05], parfois en le formulant en terme d'analyse d'atteignabilité des CFL (*Context-Free Languages*) [ZR08]. Plus récemment, [TLM+21] augmente la précision de l'analyse en utilisant la sensibilité au contexte de façon sélective. Parmi les autres approches, il faut également citer les analyses de formes (*shape analyses*) [RS01, JLRS04] utilisant la théorie de l'interprétation abstraite et qui ont pour but de faire une analyse de pointeur très précise [Min06]. Plus généralement, quand on veut faire une analyse statique d'un programme, il est judicieux, dans un certain nombre de cas, de faire une analyse de pointeur précise, car la précision supplémentaire apportée par celle-ci simplifie ensuite l'analyse

```
int* jfla (int *fst, int *snd, int **i1, int **i2, int bo) {
  *snd = *fst;
  if (bo) { fst = *i1; return *i2; } else { fst = *i2; return *i1; }
}
void main(void) {
  int u = 11, v = 12, t[3] = {0,1,2};
  int *a = &t[1], *b = &u, *c = &v;
  int **x = &a, **y = &b, **z = &c;
  struct str_t {int *fst; int *snd; } s = { c , t }, *s1 = &s, *s2 = &s;
  c = jfla(s1->fst, s1->snd, x, y, 0);
  a = jfla(s2->fst, s2->snd, y, z, 1);
}
```

Figure 1. Exemple introductif

statique du programme qui est alors plus précise et moins coûteuse que si on s'était contenté d'une analyse de pointeur imprécise [CLHY05]. C'est pourquoi la plupart des travaux récents s'orientent vers ces analyses de pointeurs très précises, malgré la difficulté théorique du problème : [Hor97] a en effet démontré que l'analyse précise de may-alias est NP-Hard (même pour une analyse *flow-insensitive*). Cependant, on peut parfois se contenter d'analyses plus légères. C'est ce que propose par exemple l'outil RTC [MVT+16] ; nous avons le même but, mais nous avons implémenté une méthode différente dans Frama-C avec le greffon Alias.

Plan La Section 2 introduit un exemple présentant notre approche. La Section 3 présente l'algorithme de Steensgaard sur lequel repose notre travail. La Section 4 est le cœur de l'article décrivant Alias, la nouvelle analyse d'alias légère de Frama-C. La Section 5 explicite nos résultats expérimentaux. Dans la Section 6, nous discutons quelques applications concrètes envisagées pour cette analyse. Enfin, la Section 7 conclut et propose quelques perspectives.

2 Exemple introductif

Pour illustrer notre méthode, voici un exemple de programme écrit en langage C (Figure 1). Ce programme commence par déclarer dans la fonction `main` deux variables entières `u` et `v`, un tableau d'entier `t`, puis trois pointeurs `a`, `b` et `c` sur ces valeurs. En particulier, `a` pointe vers la seconde case, d'indice `1`, du tableau, qui contient initialement la valeur `1`. Ensuite, on déclare des pointeurs `x`, `y` et `z` qui pointent vers les adresses auxquelles `a`, `b` et `c` sont stockées en mémoire, ainsi que deux pointeurs `s1` et `s2` vers l'adresse d'une même structure `s`.

Enfin, le programme principal fait appel à la fonction `jfla` à deux reprises. Cette fonction `jfla` prend comme arguments deux pointeurs d'entiers `fst` et `snd`, deux pointeurs de pointeurs `i1` et `i2`, et un booléen codé par un entier `bo`. Elle commence par copier la valeur pointée par `fst` dans la variable pointée par `snd` (copie d'entier), puis, selon la valeur booléenne, copie `*i1` ou `*i2` dans `fst` (copie de pointeur), et renvoie le pointeur non copié.

Cet exemple jouet a été choisi en raison de la complexité de son graphe de pointeurs (Figure 2). Il offre en effet deux niveaux d'indirections, un tableau, une structure et deux appels de fonctions. Surtout, il présente un grand nombre d'alias. Certains sont assez évidents, comme `s1`, `s2` et `&s` qui sont tous les trois aliasés. D'autres sont moins évidents, car ils résultent de l'application de la fonction `jfla`. Par exemple, après l'instruction `c = jfla(s1->fst, s1->snd, x, y, 0)`, les pointeurs `s1->fst` et `y` sont en alias (et pointent vers la variable `u`) ; de même, les pointeurs `*x` et `c` sont aliasés (et pointent vers la case du tableau `t[1]`). Puis le second appel à la fonction `jfla` va encore modifier certains de ces alias. Comme dit en introduction, la recherche exacte de tous les alias à tous les points du programme est donc un problème complexe, qui ne peut être résolu que de manière approchée. Le but de notre analyse sera donc d'obtenir un graphe de pointeurs qui représente une sur-approximation des relations entres les différents pointeurs et permet d'en déduire

ceux qui sont potentiellement aliasés.

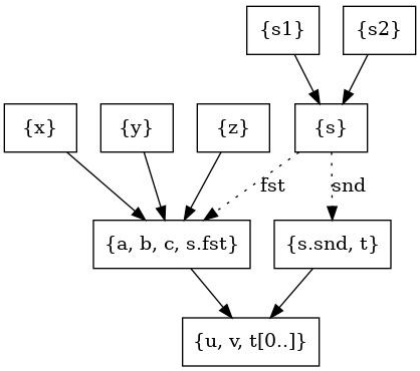


Figure 2. Graphe de pointeurs obtenu en fin d'analyse.

3 Une adaptation de l'algorithme de Steensgaard

L'analyse de pointeurs présenté dans cet article est une adaptation de l'algorithme de Steensgaard [Ste96]. C'est l'un des algorithmes les plus connus et les plus efficaces pour réaliser une analyse de *may-alias*, c'est-à-dire une analyse d'alias qui sur-approxime ses résultats, de sorte que l'ensemble concret des alias du programme est toujours inclus dans l'ensemble calculé par l'algorithme : ce critère est essentiel pour la sûreté de l'approche. Cet algorithme construit un graphe de pointeurs, c'est-à-dire un graphe orienté dont les nœuds représentent des pointeurs (ou des valeurs finales) et les arcs du type $a \to b$ signifient que le pointeur a pointe vers b. Nous rappelons ici brièvement les principes de cet algorithme.

L'article de Steensgaard [Ste96] ne parle qu'implicitement de graphe de pointeurs, car son algorithme est donné en terme d'un problème de typage et d'inférence de types. Pour le greffon Alias, cette formulation ne permettait pas de profiter pleinement des fonctionalités déjà présentes dans Frama-C, aussi nous avons choisi de reformuler cet algorithme dans une version "analyse flot de données" et de l'implémenter. C'est donc cette version que nous présentons ici.

Une analyse "flot de données" est une analyse statique qui parcourt le graphe de flot de contrôle (CFG) du programme P analysé [NNH10]. Dans notre cas, le CFG est un graphe orienté $(\mathcal{S}, \mathcal{T}, \mathcal{E}, \mathcal{I}, \mathcal{F})$, où :

— à chaque instruction de P on associe deux nœuds de $\mathcal{S}$, représentant respectivement l'état avant et après l'instruction ;
— $\mathcal{T} \subset (\mathcal{S} \times \mathcal{S})$ est l'ensemble des arcs ;
— $\mathcal{E} : \mathcal{T} \mapsto \{lv := e \mid ?(e) \mid \text{skip} \mid \texttt{return}(e)\}$ associe une étiquette à chaque arc ;
— $\mathcal{I} \in \mathcal{S}$ représente le nœud initial ;
— $\mathcal{F} \in \mathcal{S}$ représente le nœud final.

Pour les étiquettes des arcs, $lv := e$ représente une affectation dans laquelle e désigne n'importe quelle expression du langage C (y compris des appels de fonctions) et lv est une *valeur-gauche* (*lval*) représentant une zone mémoire dans laquelle on peut écrire (typiquement, une variable). $?(e)$ représente une condition : l'arc n'est franchissable que si la condition booléenne e est vraie, c'est-à-dire est un entier différent de 0 en C, skip ne fait aucune action, et $\texttt{return}(e)$ représente un retour d'une fonction renvoyant la valeur de l'expression e.

Le CFG d'un programme peut être calculé automatiquement à partir du code source, par exemple en utilisant Frama-C [BBB+21]. Frama-C effectue en outre certaines normalisations avant de le calculer, garantissant en particulier que, pour toute fonction f, il n'y a qu'un `return(__retres)`, systématiquement placé en dernière instruction de f [1]. Ainsi, la Figure 3

1. Dans le cas où la fonction retourne `void`, l'instruction est en réalité `return ;`, mais ce détail est omis dans cet article pour simplifier la présentation.

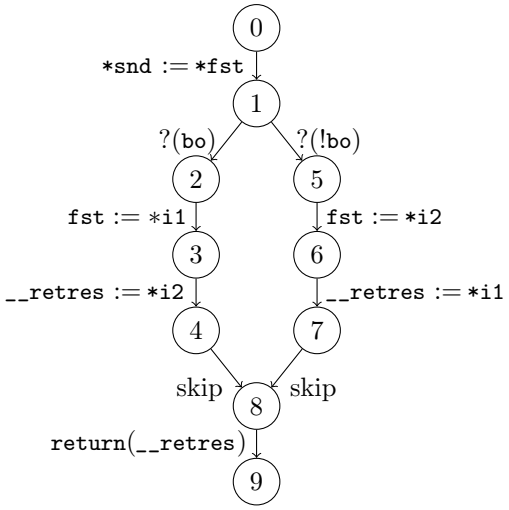


Figure 3. CFG de la fonction `jfla`.

donne le CFG de la fonction `jfla` de l'exemple introductif (cf Section 2).

Une analyse flot de données se caractérise par son sens (avant ou arrière), sa fonction de transfert (comment, à partir d'un état de l'analyseur et d'une transition du CFG, calculer l'état suivant ou précédent) et la manière de fusionner ses états dans le cas où plusieurs arcs arrivent dans un même nœud. Dans le cas de l'algorithme de Steensgaard, les états sont des graphes de pointeurs, c'est-à-dire des graphes non étiquetés ayant les deux propriétés suivantes :

1. Les nœuds du graphe de pointeurs contiennent des ensembles de *lval*. Chaque *lval* apparait dans au plus un seul nœud. De fait, ces nœuds représentent des *classes d'équivalences* de *lval* potentiellement, du fait des sur-approximations, en alias les unes avec les autres, qui sont progressivement fusionnées.
2. Chaque nœud a au plus un successeur. En effet, l'algorithme détaillé ci-dessous assure que lors des fusions, tous les successeurs potentiels sont fusionnés en un nœud unique, et ce récursivement.

Par conséquent, on définit les fonctions suivantes sur un graphe de pointeurs (G' désigne le graphe modifié à partir de G).

— **find_or_create**$(lv, G) = n, G'$: renvoie le nœud n de G contenant lv (s'il existe) ; sinon le crée, ce qui donne le graphe G'. Une extension de cette fonction à n'importe quelle expression e du langage C sera détaillée en Section 4.2.

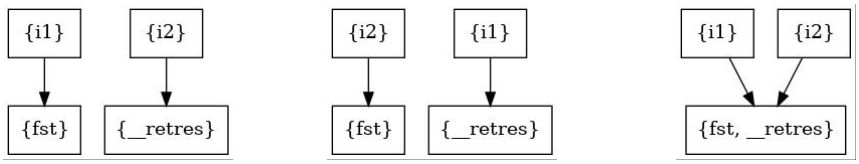
— **fusion_rec**$(n_1, n_2, G) = n, G'$: fusionne récursivement n_1 et n_2 en un nœud n ; si n_1 et/ou n_2 ont des successeurs, les fusionnent récursivement en un nœud n' (qui sera donc l'unique successeur de n).

Avec ces fonctions, nous pouvons caractériser l'analyse flot de donnée qui implémente l'algorithme de Steensgaard. C'est une analyse en avant dont l'état initial est le graphe vide. La fusion des états est une union des graphes de pointeurs, de façon à toujours sur-approximer. Pour une transition $t \in \mathcal{T}$ donnée, la fonction de transition $F_t(G)$ est calculée de la façon suivante.

— Si $\mathcal{E}(t) = \text{skip}$, alors $F_t(G) = G$ (fonction identité).

— Si $\mathcal{E}(t) = lv := e$, alors $F_t(G)$ est donné par l'Algorithme 1 ; dans cet algorithme, **is_pointer**(lv) vérifie que la *lval* lv est bien un pointeur. En pratique, Frama-C fournit cette information.

— Si $\mathcal{E}(t) = ?(e)$, alors $F_t(G) = G$ sauf si l'expression e est la constante 0, auquel cas $F_t(G) = \bot$, ce qui signifie que la branche du CFG qui suit cette transition est inatteignable.



(a) Nœud 4. **(b)** Nœud 7. **(c)** Nœuds 8 et 9.

Figure 4. Graphes calculés pour différents nœuds du CFG.

— Si $\mathcal{E}(t) = \texttt{return}(e)$, alors $F_t(G) = G$. En outre, le graphe G est enregistré comme résumé de la fonction (qui sera utilisé lors de l'appel de la fonction, cf Section 4.3).

Algorithme 1 $F_t(G)$ pour une affectation $lv := e$

```
function: fonction_transfert(lv, e, G)
  if is_pointer(lv) then
    let n1, G := find_or_create(lv, G) in
    let n2, G := find_or_create(e, G) in
    let n, G := fusion_rec(n1, n2, G) in
    return G
  else
    return G
```

L'Algorithme 1 présente deux simplifications par rapport à l'algorithme originel. En effet, pour une affectation scalaire (c'est-à-dire lorsque la variable affectée n'est pas un pointeur), il n'y a pas lieu de procéder à la fusion de nœuds dans le graphe de pointeurs. Cette simplification n'est pas possible dans l'algorithme de Steensgaard originel dans la mesure où l'information de scalarité n'est pas directement présente dans l'état de l'analyseur, c'est-à-dire le graphe de pointeurs : on sait uniquement que les nœuds avec successeurs représentent des pointeurs. Ceux sans successeurs représentent potentiellement des scalaires, mais on ne peut en être certains en cours d'analyse, tant que les fusions ne sont pas toutes effectuées. Dans notre cas, Frama-C nous fournit gratuitement, et instantanément, cette information.

L'autre simplification est que l'algorithme originel distingue plusieurs types d'affectations, tandis que notre généralisation de la fonction **find_or_create** est applicable à n'importe quelle expression du langage C : la distinction de cas est opérée au moment d'un appel à cette fonction, comme expliqué plus en détail en Section 4.2.

Appliquons donc notre algorithme de Steensgaard modifié à la fonction `jfla` dont le CFG est donné Figure 3. Au nœud 0, le graphe est vide. La première affectation est une affectation scalaire, et ne modifie pas le graphe, tout comme les conditions : le graphe est donc toujours vide aux nœuds 1, 2 et 5. Puis, les deux affectations $fst := *i1$ et `__retres` $:= *i2$ introduisent de nouveaux nœuds et arcs : la Figure 4a représente le graphe de pointeurs au nœud 4. D'une manière similaire, on obtient la Figure 4b au nœud 7. Enfin, une union des deux graphes est réalisée pour obtenir le graphe final, présenté Figure 4c ; l'algorithme réalisant cet union sera expliqué en Section 4.3.

L'algorithme de Steensgaard a donc de nombreux avantages, ce qui explique son efficacité. D'abord, le graphe est construit de façon paresseuse, un nœud n'y apparaissant que s'il représente un pointeur pour lequel il existe une information d'alias le concernant. Ensuite, il fonctionne par fusion de classes d'équivalence et permet donc d'utiliser des structures de données très efficaces, comme *Union-find* [GF64]. Enfin, l'algorithme assure que chaque nœud a au plus un arc, ce qui permet une exploration linéaire du graphe.

4 Alias, une nouvelle analyse d'alias légère pour Frama-C

Cette section présente le cœur de notre contribution. Nous détaillons d'abord en Section 4.1 le cahier des charges que notre analyse doit satisfaire. Ensuite, la Section 4.2 présente quelques détails sur la façon dont les valeurs gauches et les expressions sont globalement prises en compte. Puis, la Section 4.3 présente la manière dont nous gérons les appels de fonction. Enfin, nous discutons de quelques détails d'implémentation dans la Section 4.4.

4.1 Cahier des charges

Comme nous l'avons dit en introduction, nous voulons implémenter une analyse d'alias « légère » dans Frama-C, d'une manière indépendante à l'analyse de valeur Eva déjà existante de façon, notamment, à être plus efficace sans nécessiter de paramétrages particuliers à spécifier par un expert en analyse de code, tout en gardant une précision raisonable.

Pour être précis, nous souhaitons avoir une analyse sensible au champ (*field-sensitive analysis*), c'est-à-dire que les champs de structures ne doivent pas être systématiquement fusionnés, de façon à pouvoir spécifier qu'on est en alias avec le champ `a` mais pas avec le champ `b` d'une structure donnée, et sensible au contexte (*context-sensitive analysis*) : le contexte des appels de fonctions doit être pris en compte.

Néanmoins, pour gagner en efficacité, nous sommes prêts à certains sacrifices. En particulier, l'analyse est insensible au tableau (*array-insensitive*) et, plus généralement, à l'arithmétique de pointeurs : deux cases d'un même tableau et, plus généralement, deux pointeurs p et $p \pm i$ ($i \in \mathbb{N}$) sont systématiquement fusionnés dans la même classe d'équivalence.

Enfin, la version courante du prototype ne supportent pas certaines constructions du langage C, dont l'étude est laissée pour des travaux ultérieurs. En particulier, les conversions de types (*cast*) hétérogènes, c'est-à-dire changeant le nombre de déréférencements nécessaires pour atteindre le scalaire final, ne sont, pour l'instant, pas supportées, par exemple celles convertissant un pointeur de type `int*` en `int**` ou encore un scalaire en un pointeur ; pas plus que les fonctions récursives et les types unions. Nous allons maintenant détailler les différents algorithmes que nous avons employés.

4.2 Valeurs gauches et expressions généralisées

Représentation des valeurs gauches L'algorithme de Steensgaard ne considérait que deux types d'objets pouvant se trouver à gauche d'une affection : une variable (`x=e`) ou un pointeur déréférencé (`*x=e`). Néanmoins, le langage C autorise d'autres sortes de valeurs gauches (*lval*). En particulier, une *lval* peut désigner le champ d'une structure ou une case d'un tableau. Plus généralement, une *lval* peut être modélisée par un couple (h, o) composé d'un hôte h (soit le nom d'une variable, soit un emplacement mémoire) et d'un décalage (*offset*) o qui peut être :

— None : pas de décalage supplémentaire ;
— Field(f,o) : accès à un champ f d'une structure, précédé d'un autre décalage o ;
— Index(i,o) : accès à la case d'indice i d'un tableau, précédé d'un autre décalage o.

Cette définition est récursive, car tableaux et structures peuvent être imbriqués. Par exemple, l'expression `t[2].fst` désigne le champ `fst` de la structure se trouvant dans la case d'indice `2` du tableau `t`, ce qui est représenté par le couple hôte-offset $(\texttt{t}, \text{Field}(\texttt{fst}, \text{Index}(2, \text{None})))$. Cette représentation correspond à celle utilisée par Frama-C.

Généraliser ainsi les valeurs gauches complexifie le graphe de pointeurs et la recherche d'alias. Pour reprendre l'exemple introductif, `s1->fst` et `s2->fst` désignent le même champ `fst` de la structure `s`, puisque `s1` et `s2` sont en alias. Il faut donc ajouter au graphe de pointeurs les informations sur les décalages en plus d'avoir des nœuds pour les hôtes.

Tableaux et arithmétique de pointeurs Pour les décalages de type $Index(i,o)$, il suffit d'ajouter un arc supplémentaire au graphe du fait de vouloir une analyse insensible au décalage, car `t[i]` est alors équivalent à `*(t+i)` du point de vue de l'analyse d'alias.

Plus précisément, tout décalage `t[i]` sera considéré comme un décalage vers la première case du tableau `t[0]`, et sera donc traité dans le graphe de pointeurs comme un simple arc du nœud contenant `t` vers le nœud contenant `t[0..]`. Cette dernière notation est inspirée d'ACSL [BCF+], le langage de spécification formelle de Frama-C. Elle permet d'expliciter le fait qu'un tel nœud représente l'ensemble des cases du tableau `t`, dont la taille n'est pas nécessairement statiquement connue en C.

Le fait d'être insensible à l'arithmétique de pointeurs et au décalage lors d'un accès dans un tableau rend l'analyse plus imprécise, mais nous permet de ne pas avoir à dépendre d'une analyse de valeurs sur-approximant ces calculs arithmétiques, ce qui rend l'analyse beaucoup plus efficace.

Structures Pour les décalages de type $Field(f,o)$, que nous souhaitons conserver dans l'état de l'analyse pour être sensible aux structures, on peut soit les coder dans le graphe par des arcs spéciaux $n_1 \overset{f}{\dashrightarrow} n_2$, soit, d'une manière équivalente, les conserver à part, par exemple dans une table modifiée en même temps que le graphe. Nous supposons par la suite la première représentation (ajout d'un arc spécial). En outre, lors d'une fusion de deux nœuds n_1 et n_3 *via* la fonction **fusion_rec**(n_1,n_3,G) introduite en Section 2, et que $n_1 \overset{f}{\dashrightarrow} n_2$ et $n_3 \overset{f}{\dashrightarrow} n_4$, il faut aussi fusionner récursivement n_2 et n_4 pour préserver l'invariant que chaque nœud n'a au plus qu'un seul successeur. Ici, le nœud résultant de la fusion de n_1 et n_3 sera suivi du nœud issu de la fusion de n_2 et n_4 *via* un arc $\overset{f}{\dashrightarrow}$.

Fonction find_or_create généralisée Comme déjà expliqué, dans l'algorithme de Steensgaard originel [Ste96], chaque nœud du graphe de pointeurs contient un ensemble de variables équivalentes, c'est-à-dire possiblement en alias. La fonction **find_or_create**, présentée en Section 3, a pour but d'associer un nœud du graphe à une expression, ou de créer un tel nœud s'il n'existe pas encore. Les expressions du langage C, même limitées à celles supportées ici, sont nettement plus expressives que celles considérées par Steensgaard. La fonction **find_or_create** doit donc être adaptée. En particulier, l'expression e en argument, est simplifiée de la façon suivante : toute valeur scalaire est ignorée ou traitée comme zéro ; toute arthmétique de pointeurs est simplifiée pour ne retenir que le pointeur de base : par exemple, si `p` est un pointeur, l'expression `*(p+4)` sera simplifiée en `*(p+0)` = `*p` ; toute conversion de type (cast) est considérée comme idempotente, ce qui est possible tant que les conversions hétérogènes ne sont pas supportées. Cette procédure de simplification, notée **simplify**(e), permet de garantir qu'une expression est soit une *lval* Lv(lv), soit une addresse Addr(lv) d'une *lval* lv. Par conséquent, **find_or_create_expr**(e,G) doit chercher récursivement le nœud correspondant à **simplify**(e) dans G, et le crée s'il n'existe pas encore. L'Algorithme 2 présente cette recherche sous la forme d'un ensemble de fonctions mutuellement récursives, du fait de la structure inductive de notre représentation des valeurs gauches. Il fait appel aux fonctions auxiliaires suivantes :

- **find_or_create_var**(v,G) : trouve le nœud correspondant à la variable v (et ajoute la variable v à l'ensemble des *lval* de ce nœud), ou le crée s'il n'existe pas encore ;
- **add_edge**(n, n', G) ajoute un arc de n à n' ;
- **add_field**(n,f,n',G) ajoute l'arc spécial $n \overset{f}{\dashrightarrow} n'$ à G.

4.3 Inter-procédurabilité

Jusqu'à présent, nous avons parlé d'une analyse de pointeurs intra-procédurale, c'est-à-dire à l'intérieur d'une même fonction. La gestion de l'inter-procéduralité est souvent technique en analyse de code et cette section explique notre façon de procéder pour Alias.

Le corps de la fonction est analysé comme précédemment expliqué. Lorsque l'on atteint l'instruction `return`(`__retres`), le graphe de pointeurs obtenu est transformé en un *résumé* pour cette fonction. Pour cela, nous filtrons du graphe toutes les variables locales pour ne

Algorithme 2 Les différentes fonctions **find_or_create**

```
function: find_or_create_expr(e, G)
  match simplify(e) :
    | Lv(lv) : return find_or_create_lv(lv, G)
    | Addr(lv) :
      let n2, G := find_or_create_lv(lv, G) in
      let n1 := new_node() in
      return n1, add_edge(n1, n2, G)

function: find_or_create_lv(lv, G)
  let h, o := lv in
  let n, G := find_or_create_host(h, G) in
  return find_or_create_offset(n, o, G)

function: find_or_create_host(h, G)
  match h :
    | Var(v) : return find_or_create_var(v, G)
    | Mem(e) :
      let n1, G := find_or_create_expr(e, G) in
      if ∃(n1 → n2) ∈ G then return n2, G
      else
        let n2 := new_node() in
        return n2, add_edge(n1, n2, G)

function: find_or_create_offset(n, o, G)
  match o :
    | None : return n, G
    | Field(f, o') :
      if ∃(n -f-> n') ∈ G then return find_or_create_offset(n', o', G)
      else
        let n' := new_node() in
        return find_or_create_offset(n', o', add_field(n, f, n', G))
    | Index(0, o') :
      if ∃(n → n') ∈ G then return find_or_create_offset(n', o', G)
      else
        let n' := new_node() in
        return find_or_create_offset(n', o', add_edge(n, n', G))
```

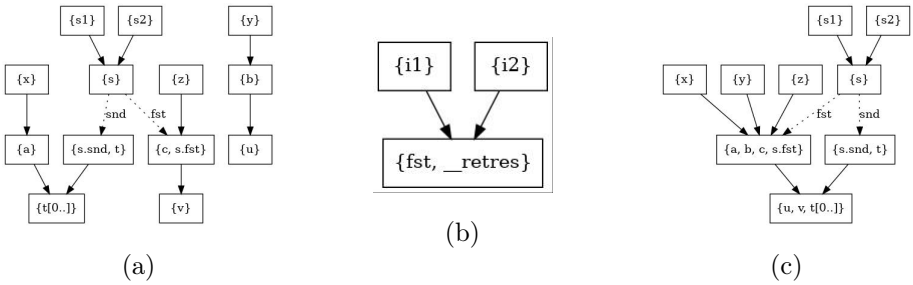


Figure 5. Graphe de pointeurs avant (a) et après (c) l'appel à la fonction `jfla` de résumé (b).

conserver que les variables globales et ses paramètres formels, ainsi que la variable distinguée `__retres`.

Lors de l'appel d'une fonction, le résumé est inséré dans le graphe de la fonction appelée en unifiant chaque paramètre formel avec l'argument réel correspondant. De même, le nœud correspondant à la (classe d'équivalence de la) variable `__retres` est unifié avec le nœud de la *lval* affectée lorsqu'elle existe. Les nœuds contenant les variables globales sont eux aussi unifiés avec ceux déjà existants dans le graphe du site d'appel.

Plus précisément, pour une fonction g avec un graphe G, appelant une fonction f :

— la normalisation du code par Frama-C assure que chaque appel de la fonction f est fait par une instruction spéciale Call(r,f,$args$) où r est la *lval* optionnelle dans laquelle est stocké le résultat de f et $args$ est la liste des arguments de la fonction f ;

— s'il n'existe pas encore un résumé R pour la fonction f, le corps de cette fonction est alors analysé s'il existe et ce résumé est créé ;

— pour chaque argument arg_i de la liste $args$, on identifie le nœud correspondant dans G grâce à la fonction **find_or_create_expr**, et on le fusionne avec celui correspondant à l'argument formel dans R ;

— on procède de même pour la *lval* r lorsqu'elle existe et le nœud correspondant à la variable `__retres` dans R.

La principale différence avec la fusion récursive **fusion_rec** employée dans l'Algorithme 1 est que ces fusions sont effectuées simultanément et qu'elles peuvent interférer entre elles, certains nœuds pouvant être partagés. En outre, les fusions faites lors d'un appel de fonction sont elles aussi récursives pour préserver l'invariant d'au plus un seul arc sortant pour chaque nœud. Ainsi, le premier appel à la fonction `jfla` donne lieu à la fusion des graphes telle que le montre la Figure 5.

4.4 Implémentation

Comme tous les autres greffons de Frama-C, Alias est écrit en langage OCaml. Il utilise les fonctionalités offertes par le noyau de Frama-C. En particulier, il utilise le module **Dataflow** qui implémente une analyse "flot de données" générique, instantiable grâce à un foncteur. Ceci permet de se concentrer sur la définition des éléments propres à notre analyse, notamment son état, c'est-à-dire ici le graphe de pointeurs, et les opérations associées, comme l'union de deux graphes, ainsi que les fonctions de transfert. L'implémentation précise de ses éléments sur un graphe de flot de contrôle, comme par exemple le calcul de point-fixe en présence de cycle, est laissé à la charge du moteur de **Dataflow**.

Alias peut être appelé par d'autres greffons Frama-C grâce à son API. Cette dernière offre à l'utilisateur diverses fonctions et itérateurs, dont :

— **fold_aliases_stmt**(f, acc, kf, s, lv) itère la fonction OCaml f à partir de l'accumulateur acc sur les alias de la *lval* lv **avant** l'instruction s de la fonction C kf

— **fold_new_aliases_stmt**(f, acc, kf, s, lv) itère la fonction f à partir de l'accumulateur acc sur les alias de la *lval* lv **après** l'instruction s de la fonction C kf

— **fold_aliases_kf**(f, acc, kf, lv) itère la fonction f à partir de l'accumulateur acc sur les alias de la *lval* lv juste avant l'instruction `return(__retres)` de la fonction C kf
— **are_aliases_stmt**(kf, s, lv_1, lv_2) renvoie vrai si et seulement si les deux *lval* lv_1 et lv_2 sont en alias avant l'instruction s de la fonction C kf
— **fold_vertex_closure**(f, acc, kf, s, lv) itère récursivement f à partir de l'accumulateur acc sur toutes les *lval* contenues dans le nœud i et, récursivement, dans tous ses successeurs, du graphe de pointeurs avant l'instruction s de la fonction C kf
— **get_state_before_stmt**(kf, s) offre un accès direct à l'état de l'analyse, c'est-à-dire à la totalité du graphe de pointeurs et des informations associées, avant l'instruction s de la fonction C kf. Cette fonctionnalité permet aux développeurs audacieux de coder des opérations complexes de bas niveau, sans exposer dans l'API des opérations inutiles pour la quasi-totalité des développeurs.

L'analyse est faite une seule fois : les résultats sont stockés de manière persistente et peuvent ensuite être utilisés via l'API autant de fois que nécessaire. Cette fonctionalité est classique dans Frama-C et offre un confort appréciable. Elle contraint néanmoins certains choix d'implémentation. Nous avons implémenté la structure de graphe de pointeurs grâce à la bibliothèque **OCamlGraph** [CFS05, CFS07]. Cette bibliothèque offre à la fois la possibilité des structures de graphes persistantes ou non-persistantes. Nous avons choisi les premières car cela nous permet de conserver un graphe en chaque point du programme, ce qui est requis par l'API ci-dessus, sans avoir besoin de copier le graphe, en temps linéaire, à chaque instruction le modifiant. Parmi les autres choix effectués, nous avons choisi de continuer l'analyse en émettant un avertissement lorsqu'une construction non supportée est rencontrée. Cela permet de passer à l'échelle, au prix d'une perte potentielle de correction, car le comportement par défaut choisi peut ignorer un certain nombre d'alias. Par exemple, lorsqu'une fonction est déclarée mais pas définie, nous supposons implicitement que cet appel ne modifie pas le graphe de pointeurs. À terme, nous pourrions nous servir des contrats ACSL pour raffiner ce comportement, mais cela ne garantirait pas pour autant la correction. C'est également le cas du code assembleur. Aussi, les conversions de type (*cast*) hétérogènes sont ignorées.

Pour finir, nous signalons qu'Alias sera intégré à la prochaine distribution publique de Frama-C (version Frama-C 28.0 (Nickel)), mais est d'ores et déjà accessibles à partir du répertoire de développement public de Frama-C [2]. Il est distribué sous licence libre LGPLv2.1.

5 Évaluation expérimentale

Comme le greffon Alias a été conçu comme une alternative à Eva, nous évaluons sa performance et son exactitude en le comparant à Eva. À cette fin, nous avons effectué des expérimentations sur une collection d'exemples, nommée Open source case studies [3] (OSCS). Cette collection contient 36 projets open-source de tailles diverses comprenant au total environ 270 000 lignes de code C, déjà préparés pour pouvoir être facilement traités avec Frama-C en général et Eva en particulier.

5.1 Performance

Pour chaque projet dans OSCS nous avons mesuré le temps d'exécution et le pic de consommation de mémoire en utilisant trois modes opératoires différents : d'abord, avec le greffon Eva, donnant le temps d'exécution t_e et la mémoire consommée m_e ; ensuite, avec le greffon Alias, donnant t_a et m_a ; enfin, sans aucun greffon, donnant t_k et m_k. Ce dernier mode nous permet de déterminer le temps et la mémoire consommé par le noyau de Frama-C pour calculer son AST. Ce prétraitement est une précondition à toutes analyses, notamment Eva et Alias. Ce temps et la mémoire nécessaire sont donc inclus dans les mesures ci-dessus.

2. `https://git.frama-c.com/pub/frama-c/-/tree/master/src/plugins/alias`
3. `https://git.frama-c.com/pub/open-source-case-studies`

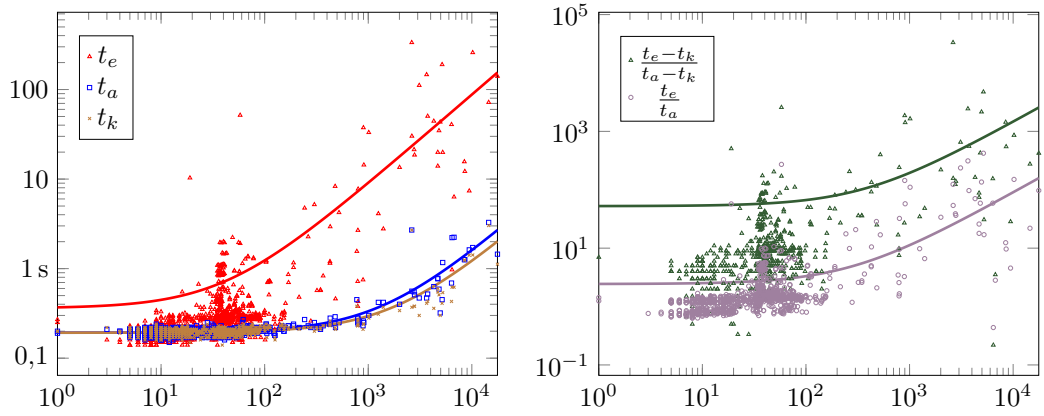


Figure 6. Indicateurs de performance en fonction du nombre de lignes de code des sous-projets ; à gauche : temps d'exécution des différentes analyses (noyau en marron, Alias en bleu, Eva en rouge) ; à droite : l'accélération d'Alias par rapport à Eva, une fois en prenant en compte t_k (vert) et une fois non (violet).

Comme t_e et t_a incluent tous les deux le temps de prétraitement t_k par le noyau de Frama-C, nous soustrayons t_k pour obtenir le seul temps d'analyse, sans prétraitement. Ainsi, nous pouvons calculer le facteur d'accélération d'Alias par rapport à Eva avec la formule $\frac{t_e-t_k}{t_a-t_k}$. Pour l'empreinte mémoire, ce genre de soustraction n'est pas pertinent. En effet, nous ne calculons qu'un *pic* de consommation de mémoire, sans savoir où il se situe précisément. En outre, il est très probable qu'il soit dans les analyseurs et non dans le noyau [4]. Pour le *facteur de réduction de l'empreinte mémoire* d'Alias par rapport à Eva, nous nous contentons donc de l'approximation $\frac{m_e}{m_a}$.

Les résultats sur OSCS sont présentés dans les figures 6 et 7, en fonction du nombre de lignes de code. Notez que quelque projets d'OSCS sont découpés en sous-projets car ils s'agit en réalité de *benchmarks* incluant eux-mêmes plusieurs programmes, donnant un nombre de points de données bien supérieur à 36. L'ordinateur avec lequel le test de performance à été effectué a comme processeur un Intel Core i7-8700 avec 3,2 GHz et 16 Go de RAM.

En totalisant toutes les valeurs pour tous les projets on obtient une accélération moyenne de 256 et en se limitant aux sous-projets de plus de mille lignes de code 235 :

$$\frac{\sum t_e - \sum t_k}{\sum t_a - \sum t_k} \approx 256 \qquad \frac{\sum_{>999} t_e - \sum_{>999} t_k}{\sum_{>999} t_a - \sum_{>999} t_k} \approx 235$$

Comme ce n'est pas pertinent d'additionner des pics de consommation de mémoire nous considérons seulement la moyenne de la réduction d'empreinte de mémoire, qui est d'environ 1,5. En se limitant aux sous-projets de plus de mille lignes de code, la moyenne est de 3,6 :

$$moyenne(\frac{m_e}{m_a}) \approx 1{,}5 \qquad moyenne_{>999}(\frac{m_e}{m_a}) \approx 3{,}6$$

5.2 Exactitude et complétude

Nous nous intéressons à la question de savoir à quel point l'ensemble d'alias A calculé par Alias est une bonne approximation du *vrai ensemble d'alias* V. Comme l'algorithme de Steensgaard calcule une sur-approximation de V, on pourrait définir l'*exactitude* d'Alias comme $\frac{|V|}{|A|}$: quel taux d'alias calculés par Alias sont de vrais alias ?

Néanmoins, comme l'implémentation actuelle d'Alias est un prototype ne traitant pas encore certains constructions syntaxiques, il peut également manquer de vrais alias dans A. Cela fausse la pertinence de $\frac{|V|}{|A|}$, car alors, plus le nombre d'éléments manquants serait

4. Dans le cas contraire, on aurait $m_a \approx m_k$ et nous obtiendrions des valeurs démesurées pour $\frac{m_e-m_k}{m_a-m_k}$.

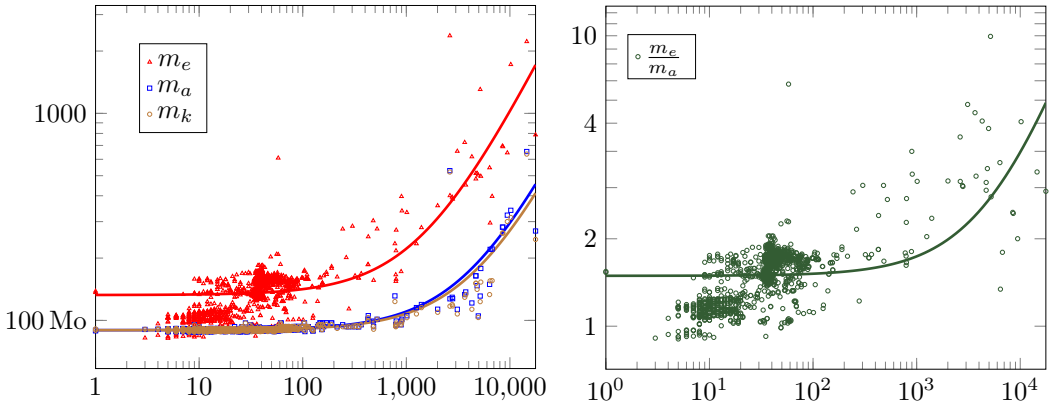


Figure 7. Indicateurs de performance en fonction du nombre de lignes de code des sous-projets ; à gauche : pics de consommation de mémoire en Mo ; à droite : facteurs d'épargne de mémoire d'Alias par rapport à Eva

grand, plus l'exactitude serait élevée. Nous pouvons restaurer une notion d'exactitude viable en prenant en compte les alias manquants comme suit : $\frac{|V \cap A|}{|A|}$. Toutefois, dans le cadre de cette expérimentation, l'ensemble V est inconnu. En particulier, les projets analysés sont trop gros pour pouvoir le calculer « à la main ». Néanmoins, nous pouvons utiliser Eva pour connaître l'ensemble des alias E calculé par Eva. Comme $E \supseteq V$, nous pouvons utiliser comme approximation (et borne supérieure) pour $\frac{|V \cap A|}{|A|}$ la formule $\frac{|E \cap A|}{|A|}$. De manière analogue, on peut définir la *complétude* d'Alias comme $\frac{|V \cap A|}{|V|}$: combien de vrais alias ont été trouvés par Alias ? De nouveau nous pouvons approximer V par E en calculant $\frac{|E \cap A|}{|E|}$.

Notez quelques choix qui ont été faits pour cette évaluation expérimentale. D'une part, elle repose sur une analyse d'alias et non une analyse de pointeurs, car cette expérimentation était nettement plus facile à réaliser avec Eva. D'autre part, les alias sont comparés seulement à la fin de chaque fonction, et non pour chaque instruction. Cette évaluation a nécessité le développement d'un greffon Frama-C dédié calculant les ensembles A et E pour toutes les *lval* du programme, à la fin de chaque fonction. La figure 8 présente les résultats d'exactitude et de complétude d'Alias pour tous les projets d'OSCS. En totalisant toutes les valeurs sur tous les projets, on obtient les résultats globaux suivants :

$$|E| = 2058 \qquad |A| = 11774 \qquad |E \cap A| = 1240 \qquad \frac{|E \cap A|}{|A|} \approx 0{,}11 \qquad \frac{|E \cap A|}{|E|} \approx 0{,}6$$

$$\mathit{moyenne}\left(\frac{|E \cap A|}{|A|}\right) \approx 0{,}29 \qquad \mathit{moyenne}\left(\frac{|E \cap A|}{|E|}\right) \approx 0{,}77$$

Limitations de l'expérimentation Les résultats relativement faibles pour l'exactitude et la complétude s'expliquent par de nombreuses raisons. D'abord, on ne dispose pas des vrais ensembles d'alias pour les exemples traités. L'utilisation d'Eva comme cadre de référence mène à des chiffres imprécis pour l'exactitude et la complétude. Ensuite, grâce à la structure de certains exemples d'OSCS, Eva peut utiliser des contrats attachés aux fonctions des libraries standard pour son analyse abstraite et, certaines de ces fonctions sont mêmes traités de manières natives, comme des *built-ins*, ce qui améliore grandement leur précision (et leur efficacité). Un tel traitement n'existe pas à ce jour pour Alias. Puis, Alias fonctionne très mal sur certains exemples spécifiques, en général à cause d'une construction spécifique non encore supportée, ce qui fausse fortement le calcul des moyennes. Enfin, certaines causes peuvent être encore inconnues : pour mieux les comprendre, il faudrait scruter plus en détails les résultats exacts d'Eva et Alias sur nos exemples. Cette étude n'a pas encore été effectuée.

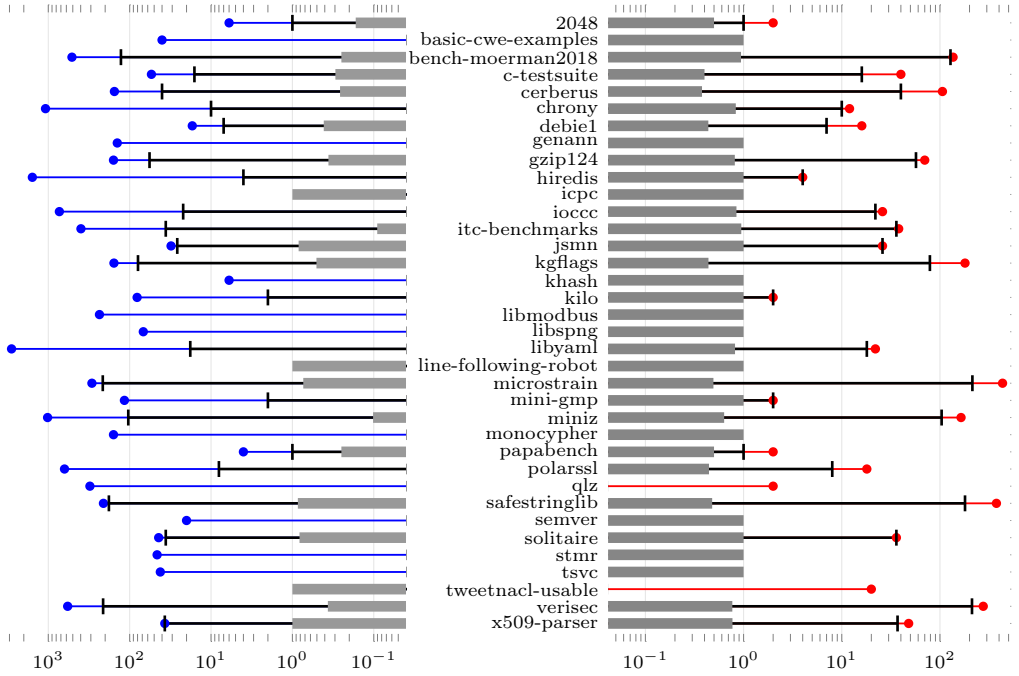


Figure 8. Indicateurs d'exactitude (à gauche) et de complétude (à droite) :

6 Applications potentielles

Cette section présente quatre applications potentielles d'Alias. En effet, même si ce nouveau greffon a été développé très récemment, plusieurs applications potentielles sont prévues ou en cours. Cette section en présente quatre, deux pour de nouveaux analyseurs et deux pour des analyseurs existants : d'une part, les Sections 6.1 et 6.2 introduisent respectivement des applications à la génération de programmes concurrents et à une analyse pour l'intégrité du flot de contrôle, tandis que les Sections 6.3 et 6.4 montrent l'intérêt d'Alias pour optimiser d'une manière correcte des traitements dans WP, le greffon de vérification déductive de Frama-C, et dans E-ACSL, celui dédié à la vérification à l'exécution. La première application, pour la concurrence, est plus détaillée que les autres.

6.1 Génération automatique de programmes concurrents

Les programmes concurrents sont, par nature, plus complexes que les programmes séquentiels et, donc, sont davantage sujets aux erreurs. Certaines classes d'erreurs leur sont propres, comme les *data races*, les *deadlocks* ou les violations de comportements attendus par l'utilisateur, *e.g.* l'atomicité d'une section de code, c'est-à-dire le fait que d'autres instructions peuvent s'exécuter en parallèle sans que leurs effets n'impactent ceux de la section considérée, ou l'ordonnancement entre instructions de tâches parallèles. Vérifier l'absence d'erreur s'avère particulièrement difficile dans ce contexte. Non seulement il est nécessaire de raisonner suffisamment finement sur le modèle mémoire sous-jacent, mais il faut aussi tenir compte de tous les entrelacements possibles entre les tâches du programme : la multitude de combinaisons possibles rend inefficace tout effort de validation à base de tests.

Certains interpréteurs abstraits, comme le greffon MThread [KKP+15] de Frama-C ou AstréeA [Min15] permettent de vérifier l'absence de *data races* et de *deadlocks* dans les programmes C concurrents. Ces approches, fondées sur une abstraction des interférences entre tâches d'un programme, présentent plusieurs inconvénients : difficulté à corriger les erreurs identifiées, impossibilité de traiter les erreurs complexes (violations d'atomicité ou

```c
int *out, in[10], x;
void * t1(void * args) {
  int *tmp1 = &x;
  for (int i = 0; i < 10; i+=2) *tmp1+=in[i];
}
void * t2(void * args) {
  int *tmp2 = &x;
  for (int i = 1; i < 10; i+=2) *tmp2+=in[i];
}
void main(void) {
  out = &x;
  *out = 0;
  _task_(t1);
  _task_(t2);
  printf ("Current output = %d\n", *out);
  _after_task_(t1,t2);
}
```

Figure 9. Exemple de programme en entrée

d'ordonnancement) et coût important de l'analyse. Les langages corrects par construction, comme Rust [YSZ19] ou Concurrent Cyclone [GPS10], garantissent par typage l'absence de *data races*. Cependant, les erreurs plus complexes ne leur sont pas accessibles. Ils sont par ailleurs difficiles à adopter dans certains contextes industriels, de part leur difficulté d'appréhension et le nombre important de programmes existants écrits dans des langages traditionnels. Autolocker [MZGB06] offre un compromis intéressant : l'outil est fondé sur un langage proche du C, avec des directives de parallélisme de haut niveau. Les programmes y sont écrits en C séquentiel et des directives permettent de déclarer le lancement de tâches ou l'atomicité de sections de code. L'outil place des verrous dans le code de manière à assurer l'atomicité et l'absence de *deadlocks*. En revanche, il est de la responsabilité de l'utilisateur de déclarer quels verrous protègent quelles variables. De plus, Autolocker ne permet pas de spécifier de propriétés d'ordonnancement temporelles entre instructions.

Nous proposons une approche plus complète, fondée sur un langage d'entrée proche d'Autolocker. Le langage d'entrée est augmenté de la possibilité de placer des directives d'ordonnancement entre des instructions à l'intérieur des tâches parallèles. La Figure 9 présente ce langage d'entrée à travers un exemple. Le programme a pour but de sommer les éléments d'un tableau. Les deux tâches `t1` et `t2` sont respectivement chargées des éléments d'indices pairs et impairs. Nous complexifions le cas d'usage en introduisant des alias entre pointeurs. La directive `_task_(t)` spécifie le lancement d'une tâche `t`. La directive `_after_task_(t)` spécifie un point d'ordonnancement après lequel les instructions doivent attendre la fin de l'exécution de la tâche `t` avant de s'exécuter.

À partir du programme en entrée, nous identifions les mécanismes de protection adéquats pour chaque instruction dans le programme. La stratégie est déployée en trois grandes étapes (identification des variables partagées, ensemble de verrous à acquérir, placement correct des verrous). Ici, nous nous intéressons uniquement à la première étape de cette stratégie. Nous assumons, en outre, certaines restrictions sur le langage C, les mêmes que les restrictions actuelles du greffon Alias (discutées en Section 4.1), et sur les directives de parallélisme : une tâche ne peut pas en lancer une autre et chaque tâche ne peut être lancée qu'une seule fois dans le programme.

Commençons par poser le problème à résoudre : pour chaque instruction du programme, nous voulons déterminer les noms de variables du programme qui pourraient référencer une zone de mémoire partagée. La présence de potentiels alias dans le programme complexifie fortement l'analyse. Le greffon Alias présenté dans les précédentes sections étant léger et ne nécessitant aucun paramétrage, il répond parfaitement à nos besoins. Prenons pour exemple

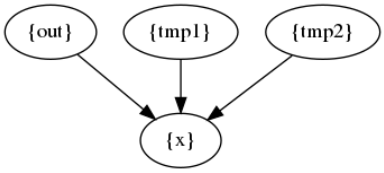


Figure 10. Graphe de pointeurs fusionné.

l'instruction `printf` à la ligne 20 du programme de la Figure 9.

Premièrement, nous identifions syntaxiquement les variables accédées, ainsi que les modes d'accès associées. Ici, il y a un seul accès en lecture, à la zone mémoire `*out`. Ensuite, nous identifions l'ensemble des tâches (et des instructions du `main` si l'instruction considérée est à l'intérieur d'une tâche) qui peuvent être exécutées en parallèle. Les éléments de cet ensemble sont appelés les compétiteurs de l'instruction. Ici, les deux compétiteurs sont `t1` et `t2`.

Nous utilisons le greffon Alias pour obtenir le graphe de pointeurs correspondant au point de programme situé avant l'exécution de l'instruction considérée. Si l'instruction fait partie de la fonction `main`, l'analyse a pour point de départ le début du programme et ne tient pas compte des tâches parallèles, sauf celles dont l'exécution est entièrement terminée. Si l'instruction fait partie du corps d'une tâche t, le point de départ de l'analyse est le début de t, dont le contexte d'appel est considéré comme vide (pas d'aliasing). Ici, au point de programme considéré, la variable `out` pointe vers `x`. Nous calculons ensuite les graphes de pointeurs après exécution des différents compétiteurs de l'instruction, considérés isolément les uns des autres. Dans le cas d'une instruction située à l'intérieur d'une tâche t, nous collectons également le graphe de pointeurs correspondant au point de programme situé avant le lancement de t. Ici, `tmp1` pointe vers `x` pour `t1` et `tmp2` pointe vers `x` pour `t2`.

Nous avons défini une notion de fusion de graphes de pointeurs, qui permet de sur-approximer les possibles zones mémoires pointées par les différentes variables en fonction de l'entrelacement considéré. Nous fusionnons le graphe collecté avant l'instruction considérée, les graphes collectés ci-dessus pour les compétiteurs et, si nécessaire, le graphe au point de lancement de la tâche dans laquelle se trouve l'instruction. La Figure 10 illustre la notion de graphe fusionné dans le cas de notre exemple. Ensuite, nous collectons les ensembles de variables syntaxiquement accédées par les différents compétiteurs, en tenant compte des modes d'accès mais en ignorant leurs compétiteurs respectifs. Ici, `t1` (resp. `t2`) accède à `tmp1` (resp. `tmp2`) en écriture et au tableau `in` en lecture. À partir de ces informations, nous identifions les variables à protéger, c'est-à-dire accédées par au moins un compétiteur, avec un accès au moins en écriture parmi l'instruction et les compétiteurs, en tenant compte de potentiels alias. Pour notre exemple, la variable `out` est identifiée car accédée en lecture dans l'instruction et en écriture par `t1` et `t2` *via* `tmp1` et `tmp2`, toutes deux en alias avec `out`.

6.2 Intégrité du flot de contrôle avec attestation à distance

L'*attestation à distance* permet la détection des comportements inattendus dans un composant logiciel. C'est une technique critique pour l'informatique de confiance, qui donne des garanties fortes concernant l'intégrité d'un composant logiciel. L'*intégrité du flot de contrôle* (*control flow integrity*, CFI) est une technique qui vérifie qu'un programme suit le flot de contrôle attendu pendant son exécution, par observation (*monitoring*). Le *flot de contrôle attendu* est une sur-approximation de tous les chemins d'exécution possible d'un programme qui s'exécute correctement. Il est déterminé par une analyse statique qui, en présence de pointeurs de fonction, dépend d'une analyse de pointeurs. Des déviations du flot de contrôle par rapport à celui attendu peuvent indiquer des manipulations du programme par un attaquant ou des défaillances matérielles. En attestant l'intégrité du flot de contrôle, CFI permet d'augmenter la confiance dans la sécurité et la sûreté d'un logiciel.

Nous sommes en train de développer un outil logiciel permettant l'attestation à distance de l'intégrité du flot de contrôle pour des programmes implémentés en C. Les composants

de cette architecture sont les suivants.

— Une analyse statique qui extrait le graphe de flot de contrôle du programme.
— Une instrumentation de code source qui insère, à certains points de programme (par exemple, avant les appels de fonction), des *points de contrôle*. Ces points de contrôle journalisent, pendant l'exécution du programme, le comportement de ce dernier (par exemple : le fait que la fonction f a été appelée depuis une certaine position p) et les transmettent au vérificateur.
— Un *vérificateur*, c'est-à-dire un serveur qui dispose du graphe de flot de contrôle statique et qui peut attester que le comportement journalisé et transmis depuis le programme respecte bien le flot de contrôle attendu.
— Un protocole de communication sécurisée qui assure aussi l'intégrité du code ajouté par l'instrumentation en s'appuyant sur un TPM (*Trusted Platform Module*), c'est-à-dire un module de stockage matériel sécurisé.
— Une architecture de déploiement qui facilite l'ensemble des étapes nécessaires pour exécuter un programme : l'analyse et l'instrumentation du programme, l'exécution du vérificateur et la transmission du graphe de flot de contrôle au vérificateur, la compilation et l'exécution du programme instrumenté.

L'analyse et l'instrumentation ont été implémentées comme un greffon pour Frama-C, appelé rCFI, non encore publiquement disponible. Ce greffon utilise au choix Eva ou Alias comme analyse de pointeurs. L'exactitude de l'analyse de pointeurs a un impact direct sur l'exactitude du graphe de flot de contrôle généré. Néanmoins, ici, les enjeux principaux du projet sont d'une part la minimisation des surcoûts en temps d'exécution et, d'autre part, la maximisation de l'exactitude du graphe de flot de contrôle. En effet une meilleure exactitude restreint la surface d'attaque, c'est-à-dire les comportements possibles qu'un attaquant peut exploiter pour arriver à son but. De plus il faut que les coûts d'analyse restent raisonnables. Il est donc intéressant d'avoir à disposition deux analyses avec des points forts différents et de déployer l'une ou l'autre selon le cas d'utilisation.

6.3 Optimisation pour la vérification déductive

WP est le greffon dédié à la vérification déductive de Frama-C. Il prend en entrée un code C annoté avec des spécifications écrites dans un langage de spécification formelle appelé ACSL [BCF+] dans le but de démontrer que ce code satisfait les spécifications. Pour cela, il génère des propriétés, appelées obligations de preuve : si l'ensemble de ces propriétés sont démontrées, alors le code C satisfait les spécifications [HH19]. Démontrer les obligations de preuve repose sur l'usage de prouveurs automatiques comme Alt-Ergo ou d'assistants à la preuve comme Coq. Un des enjeux pour ce type d'outils consiste à avoir le taux d'automatisation le plus élevé possible, c'est-à-dire à limiter au maximum l'usage des assistants à la preuve.

Générer d'une manière correcte les obligations de preuve demande une interprétation fine des sémantiques de C et d'ACSL. Il convient en outre de définir de bonnes abstractions pour représenter les opérations de ces langages. Ces abstractions permettent notamment de représenter les objets mathématiques comme les nombres entiers ou les nombres flottants, ainsi que les objets écrits en mémoire par le programme. En particulier, pour ces derniers objets, les abstractions liées à la gestion de la mémoire, appelées modèles mémoires, sont critiques dans le cas de l'analyse d'un code C : un modèle dit « de bas niveau » permet d'exprimer finement les propriétés sur la mémoire, mais a tendance à générer des obligations de preuve difficiles, voire impossibles, à prouver automatiquement, alors qu'un modèle dit de « haut niveau » autorise un taux élevé d'automatisation mais exclut la vérification de programmes contenant certaines opérations non représentables dans le modèle. Notamment, certains modèles ne fonctionnent pas en présence d'alias dans le code C.

Un courant de recherche actuel consiste donc à définir de nouveaux modèles mémoires permettant la vérification automatique de programmes de bas niveau, notamment en présence d'alias. Les techniques actuelles reposent notamment sur de l'analyse de régions

visant à séparer la mémoire en zones disjointes (appelées régions) permettant de garantir des propriétés d'isolation inter-régions augmentant l'automatisation des preuves. Ainsi, Alias sera utilisé dans le cadre de la définition de nouveaux modèles mémoires bas niveau qui seront développés dans l'année qui vient.

6.4 Optimisation pour la vérification à l'exécution

E-ACSL est le greffon de Frama-C dédié à la vérification à l'exécution. Il prend en entrée un programme C annoté avec des annotations ACSL et génère un nouveau programme C dans lequel les annotations ont été converties en code C : l'exécution du programme est fonctionnellement équivalente au programme C d'origine lorsque toutes les annotations sont correctes, ou sinon arrête l'exécution sur la première annotation incorrecte (ce comportement est celui par défaut, mais il est modifiable) [SKV17]. L'enjeu, ici, consiste à générer un code correct et le plus efficace possible.

Le langage ACSL permet notamment à l'utilisateur d'exprimer des propriétés sur la mémoire, par exemple le fait que des accès mémoires soient valides, c'est-à-dire que le programme a le droit d'y accéder en lecture et/ou en écriture, ou sur le fait que certaines données ont été correctement initialisées. Vérifier de telles propriétés pendant l'exécution du programme est compliqué et requiert, là encore, l'utilisation d'un modèle mémoire, cette fois dynamique, c'est-à-dire évoluant en cours d'exécution [VSK17]. Son usage requiert néanmoins une instrumentation invasive du code, requérant notamment d'ajouter de nouvelles instructions pour toutes les écritures mémoires de façon à mettre à jour le modèle mémoire d'E-ACSL. Or, si les annotations à vérifier sont indépendantes de certaines écritures mémoires, l'instrumentation de ces dernières est en réalité inutile. Ainsi, une analyse statique dédiée a été développée dans E-ACSL permettant de calculer une sur-approximation correcte des zones mémoires à surveiller [LKLS18], ce qui permet d'améliorer grandement l'efficacité du code produit [JKS16].

Cette analyse statique repose en réalité sur une analyse de pointeurs qui est, dans l'implémentation actuelle d'E-ACSL, effectuée en même temps que le reste de l'analyse. Ceci pose un certain nombre de problèmes pratiques, aussi bien au niveau du passage à l'échelle que de la correction, ce qui est problématique. Une étude formelle de cette analyse [LKLS18] a permis de démontrer qu'elle pouvait être paramétrée par l'analyse de pointeurs sous-jacente. Ainsi, nous envisageons d'implémenter une nouvelle version correcte et plus efficace de l'analyse actuelle, qui reposerait sur Alias pour effectuer son analyse de pointeurs.

7 Conclusion et perspectives

Cet article a présenté Alias, une nouvelle analyse de pointeurs fonctionnant sur du code C et implémentée comme un greffon de la plateforme Frama-C dédiée à l'analyse de code C. Il s'agit d'une adaptation de l'algorithme de Steensgaard [Ste96]. Elle est rapide et légère, dans le sens où elle demande moins de ressources (temps, mémoire) que l'analyse du greffon Eva, et ne nécessite aucun paramétrage de la part de l'utilisateur.

Cette caractéristique permet d'envisager de l'intégrer dans des analyseurs plus importants. Nous envisageons ainsi de l'utiliser dans quatre contextes différents, aussi bien liés au développement de nouveaux analyseurs pour la génération de programmes concurrents et la vérification de l'intégrité du flot de contrôle, que pour améliorer des analyseurs existants, comme WP dédié à la vérification déductive et E-ACSL dédié à la vérification à l'exécution.

L'efficacité et la précision d'Alias ont été mesurées sur un *benchmark* existant et représentatif. Les résultats démontrent qu'elle répond au besoin, en les comparant notamment à l'analyse d'Alias incluse dans le greffon Eva de Frama-C.

L'implémentation actuelle reste néanmoins un prototype. Les travaux futurs incluent donc l'extension de l'analyse pour supporter l'ensemble des constructions du langage C, par exemple les casts hétérogènes ou les types unions. Ils visent également à augmenter encore l'efficacité de l'analyseur.

Références

[And94] Lars Ole Andersen : *Program analysis and specialization for the C programming language*. Thèse de doctorat, University of Copenhagen, 1994.

[BBB+21] Patrick Baudin, François Bobot, David Bühler, Loïc Correnson, Florent Kirchner, Nikolai Kosmatov, André Maroneze, Valentin Perrelle, Virgile Prevosto, Julien Signoles et Nicky Williams : The Dogged Pursuit of Bug-Free C Programs : The Frama-C Software Analysis Platform. *Communications of the ACM*, 2021.

[BBY17] Sandrine Blazy, David Bühler et Boris Yakobowski : Structuring Abstract Interpreters through State and Value Abstractions. *In International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'17)*, janvier 2017.

[BCF+] Patrick Baudin, Pascal Cuoq, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy et Virgile Prevosto : ACSL : ANSI/ISO C Specification Language.

[CC77] Patrick Cousot et Radhia Cousot : Abstract interpretation : A unified lattice model for static analysis of programs by construction or approximation of fixpoints. *In Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '77, 1977.

[CFS05] Sylvain Conchon, Jean-Christophe Filliâtre et Julien Signoles : Le foncteur sonne toujours deux fois. *In Journées Francophones des Langages Applicatifs (JFLA)*, mars 2005.

[CFS07] Sylvain Conchon, Jean-Christophe Filliâtre et Julien Signoles : Designing a generic graph library using ML functors. *In Trends in Functional Programming (TFP)*, avril 2007.

[CLHY05] Tong Chen, Jin Lin, Wei-Chung Hsu et Pen-Chung Yew : An empirical study on the granularity of pointer analysis in C programs. *In Workshop on Languages and Compilers for Parallel Computing (LCPC)*, 2005.

[Cou22] Patrick Cousot : *Principles of Abstract Interpretation*. MIT Press, 2022.

[DMM98] Amer Diwan, Kathryn S. McKinley et J. Eliot B. Moss : Type-Based Alias Analysis. *In International Conference on Programming Languages, Design and Implementation (PLDI)*, 1998.

[GF64] Bernard A. Galler et Michael J. Fisher : An improved equivalence algorithm. *Communications of the ACM*, mai 1964.

[GPS10] Prodromos Gerakios, Nikolaos Papaspyrou et Konstantinos Sagonas : Race-free and memory-safe multithreading : Design and implementation in cyclone. *In Workshop on Types in Language Design and Implementation (TLDI)*, 2010.

[HH19] R. Hähnle et M. Huisman : *Deductive Software Verification : From Pen-and-Paper Proofs to Industrial Tools*. 2019.

[Hor97] Susan Horwitz : Precise flow-insensitive may-alias analysis is np-hard. *Transactions on Programming Languages and Systems (TOPLAS)*, 1997.

[HT01] Nevin Heintze et Olivier Tardieu : Demand-Driven Pointer Analysis. *In Conference on Programming Language Design and Implementation (PLDI)*, 2001.

[JKS16] Arvid Jakobsson, Nikolai Kosmatov et Julien Signoles : Fast as a Shadow, Expressive as a Tree : Optimized Memory Monitoring for C. *Science of Computer Programming*, octobre 2016.

[JLRS04] Bertrand Jeannet, Alexey Loginov, Thomas Reps et Mooly Sagiv : A relational approach to interprocedural shape analysis. *In International Static Analysis Symposium (SAS)*, 2004.

[KKP+15] Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles et Boris Yakobowski : Frama-c : A software analysis perspective. *Formal aspects of computing*, 2015.

[LKLS18] Dara Ly, Nikolai Kosmatov, Frédéric Loulergue et Julien Signoles : Soundness of a dataflow analysis for memory monitoring. *In Workshop on Languages and Tools for Ensuring Cyber-Resilience in Critical Software-Intensive Systems (HILT)*, novembre 2018.

[Min06] Antoine Miné : Field-sensitive value analysis of embedded c programs with union types and pointer arithmetics. *In Conference on Language, Compilers, and Tool Support for Embedded Systems (LCTES)*, 2006.

[Min15] Antoine Miné : Astréea : A static analyzer for large embedded multi-task software. *In 16th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'15)*, volume 8931, page 3. Springer, 2015.

[MVT+16] Reed Milewicz, Rajesh Vanka, James Tuck, Daniel Quinlan et Peter Pirkelbauer : Lightweight runtime checking of C programs with RTC. *Computer Languages, Systems & Structures*, 2016.

[MZGB06] Bill McCloskey, Feng Zhou, David Gay et Eric Brewer : Autolocker : synchronization inference for atomic sections. *In symposium on Principles of Programming Languages (POPL)*, 2006.

[NNH10] Flemming Nielson, Hanne R. Nielson et Chris Hankin : *Principles of Program Analysis*. Springer Publishing Company, Incorporated, 2010.

[Our15] Alain Ourghanlian : Evaluation of static analysis tools used to assess software important to nuclear power plant safety. *Nucl. Eng. Technol.*, 2015.

[RS01] Noam Rinetzky et Mooly Sagiv : Interprocedural shape analysis for recursive programs. *In International Conference on Compiler Construction (CC)*, 2001.

[SB15] Yannis Smaragdakis et George Balatsouras : Pointer Analysis. *Foundations and Trends in Programming Languages*, 2015.

[SGSB05] Manu Sridharan, Denis Gopan, Lexin Shan et Rastislav Bodík : Demand-Driven Points-to Analysis for Java. *In Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, page 59–76, New York, NY, USA, 2005. Association for Computing Machinery.

[SKV17] Julien Signoles, Nikolai Kosmatov et Kostyantyn Vorobyov : E-acsl, a runtime verification tool for safety and security of c programs. *In International Workshop on Competitions, Usability, Benchmarks, Evaluation, and Standardisation for Runtime Verification Tools (RV-CuBES)*, septembre 2017.

[Ste96] Bjarne Steensgaard : Points-to analysis in almost linear time. *In Symposium on Principles of Programming Languages (POPL)*, 1996.

[TLM+21] Tian Tan, Yue Li, Xiaoxing Ma, Chang Xu et Yannis Smaragdakis : Making Pointer Analysis More Precise by Unleashing the Power of Selective Context Sensitivity. 2021.

[VSK17] Kostyantyn Vorobyov, Julien Signoles et Nikolai Kosmatov : Shadow state encoding for efficient monitoring of block-level properties. *In International Symposium on Memory Management (ISMM)*, juin 2017.

[YSZ19] Zeming Yu, Linhai Song et Yiying Zhang : Fearless concurrency ? Understanding concurrent programming safety in real-world Rust software. *arXiv preprint arXiv :1902.01906*, 2019.

[ZR08] Xin Zheng et Radu Rugina : Demand-Driven Alias Analysis for C. *SIGPLAN Not.*, janvier 2008.