Rester statique pour devenir plus rapide, plus précis et plus mince

Arvid Jakobsson, Nikolai Kosmatov et Julien Signoles

CEA, LIST, Laboratoire de Sûreté des Logiciels
91191 Gif-sur-Yvette Cedex
Julien.Signoles@cea.fr

Résumé

E-ACSL est un greffon de Frama-C, une plateforme d'analyse de codes C qui est développée en OCaml. Son but est de transformer un programme C formellement annoté dans le langage de spécification éponyme E-ACSL en un autre programme C dont le comportement à l'exécution est équivalent si toutes les spécifications sont dynamiquement vérifiées et qui échoue sur la première spécification fausse sinon. Pour cela, une instrumentation du programme source est notamment effectuée afin, d'une part, de conserver les informations nécessaires à l'évaluation des prédicats E-ACSL et, d'autre part, de traduire correctement ces derniers.

Cet article présente deux analyses statiques qui améliorent grandement la précision de cette transformation de programmes en réduisant l'instrumentation effectuée. Ainsi, le code généré est plus rapide et consomme moins de mémoire lors de son exécution. La première analyse est un système de types fondé sur une inférence d'intervalles et permettant de distinguer les entiers (mathématiques) pouvant être convertis en des expressions C de type « entier machine » de ceux devant être traduits vers des entiers en précision arbitraire. La seconde analyse est une analyse flot de données arrière sur-approximante paramétrée par une analyse d'alias. Elle permet de limiter l'instrumentation des opérations sur la mémoire à celles ayant un impact potentiel sur la validité d'une annotation formelle.

1. Introduction

Frama-C [5, 12] est une plateforme à vocation industrielle, majoritairement développée en *OCaml*, et permettant l'analyse de codes écrits en *C*. Elle permet d'annoter formellement des programmes *C* à l'aide du langage de spécification *ACSL* et fournit des greffons effectuant différentes analyses et transformations de programmes qui peuvent collaborer entre elles. À ses origines en 2008, *Frama-C* était orienté analyse statique, mais a depuis été élargi aux analyses dynamiques.

Les techniques existantes de vérification statique de programmes, comme le typage [16], l'interprétation abstraite [4] ou la preuve de programmes [7], permettent de garantir l'absence de certaines classes d'erreurs à l'exécution (*runtime errors,* ou RTEs). Néanmoins, lorsqu'on souhaite vérifier des programmes issus du monde réel, le problème de la vérification étant indécidable, soit l'expressivité des propriétés vérifiées est limitée, soit un effort potentiellement non négligeable est demandé à l'utilisateur pour terminer le processus de vérification.

Ainsi, la vérification dynamique de programmes [2], en particulier l'observation de code (*monitoring*), constitue une alternative à la vérification statique en permettant de surmonter ces limitations, au prix d'une absence de garantie forte : seules certaines exécutions sont vérifiées, mais l'effort demandé à l'ingénieur validation est faible. Dans ce contexte, *E-ACSL* [6] est un greffon de *Frama-C* transformant un programme p annoté dans un large fragment de *ACSL* (nommé *Executable-ACSL*, ou éponymement *E-ACSL* [18]) en un autre programme p', qui a le même comportement

observationnel que p pour les exécutions ne violant aucune annotation et qui échoue sur la première annotation erronée sinon.

Étant fondé sur une logique du premier ordre typée dont les termes sont des expressions C pures et étendue avec un certain nombre de constructions dédiées, *E-ACSL* est un langage particulièrement expressif. On peut distinguer notamment deux sortes de constructions. D'une part, les constantes et les opérateurs sur les entiers sont en précision arbitraire dans $\mathbb{Z}$: les entiers ne sont pas bornés et les opérateurs mathématiques ne peuvent pas déborder. D'autre part, un ensemble de prédicats et de fonctions logiques prédéfinis permet d'exprimer des propriétés sur la mémoire. Par exemple, `\valid(p)` exprime le fait que le pointeur p pointe vers un emplacement valide de la mémoire, *i.e.* une zone correctement allouée par le programme, non encore libérée et qu'il a le droit de modifier. Alors que ces deux sortes de constructions augmentent fortement le pouvoir expressif du langage, elles complexifient la traduction effectuée par le greffon *E-ACSL* pour les vérifier à l'exécution. En effet, d'une part les opérations entières doivent être effectuées en précision arbitraire et ne peuvent être substituées par les opérateurs C correspondant qui pourraient déborder et, d'autre part, les opérations comme `\valid` requièrent une lourde instrumentation pour enregistrer les opérations effectuées sur la mémoire (allocation, dé-allocation, initialisation, ...) afin d'en déduire notamment les emplacements valides. Ainsi, *E-ACSL* génère du code qui dépend à la fois de *GMP*[1], bibliothèque C d'entiers en précision arbitraire, et d'une autre bibliothèque C dédiée modélisant la mémoire [13]. L'inconvénient de cette lourde instrumentation est que le programme ainsi généré est beaucoup plus lent et occupe beaucoup plus de mémoire que le programme initial, ou même que le programme qui serait généré en remplaçant l'utilisation de *GMP* par les opérations C correspondantes.

Cet article présente deux analyses statiques qui ont été développées dans *E-ACSL* afin de réduire le surcoût en temps et en mémoire du code généré. La première analyse est un système de (sous-) types reposant sur une inférence d'intervalles. Elle permet de remplacer les utilisations d'entiers et d'opérateurs *GMP* par les entiers machines et les opérateurs C correspondants de manière sûre, *i.e.* lorsque les entiers sont représentables en machine et que les opérations ne peuvent pas déborder. Autant l'analyse d'intervalles sur laquelle le système repose est standard, autant le fait que ce système détermine conjointement le type du résultat et le type de l'opérateur est, selon nous, plus original. La seconde analyse est une analyse flot de données arrière sur-approximante paramétrée par une analyse d'alias. Elle permet de ne pas instrumenter les opérations mémoires qui n'ont aucun impact sur les annotations *ACSL* exprimant des propriétés sur la mémoire, comme la validité d'un pointeur. Cette analyse est une analyse de type « variables vivantes » standard. Son originalité provient de la prise en compte des annotation logiques, du fait qu'elle soit paramétrée par une analyse d'alias et qu'elle calcule également la raison d'observation des variables concernées.

En section 2, nous introduisons d'abord un petit langage impératif avec annotations sur lequel notre étude théorique s'appuie. Ensuite, la section 3 détaille le système de types améliorant le support des entiers, tandis que la section 4 explicite l'analyse flot de données réduisant l'instrumentation des opérations mémoires. Enfin, la section 5 discute de l'implantation et des expérimentations effectuées.

2. Langage d'étude

Conduire une étude formelle complète sur un langage aussi complet et complexe que le C est un projet plus que conséquent (même si des travaux antérieurs, par exemple [1, 8, 10], montrent que c'est possible). Aussi nous limitons-nous ici à l'étude d'un langage impératif de taille plus modeste dont la syntaxe est présentée figure 1. Ce langage d'étude demeure néanmoins suffisamment représentatif pour nous permettre d'illustrer les difficultés principales de nos analyses.

Ce langage contient des entiers machines bornés, les opérations arithmétiques et booléennes les plus classiques ainsi que des pointeurs, avec un opérateur permettant d'obtenir l'adresse d'une variable (`&`)

1. `http://gmplib.org`

Valeurs gauches	$l ::=$	x	variable
	$\mid$	$\star a$	déréférencement d'un pointeur
Valeurs mémoires	$a ::=$	l	valeur gauche
	$\mid$	a `++` n	décalage de pointeur
	$\mid$	$\&l$	adresse
Expressions	$e ::=$	n	constante entière
	$\mid$	a	valeur mémoire
	$\mid$	e `+` $e \mid e$ `-` $e \mid e$ `*` $e \mid e$ `/` e	expressions arithmétiques
	$\mid$	e `==` $e \mid e$ `<=` $e \mid$ `!` e	expressions booléennes à valeur entière
Instructions	$i ::=$	;	sans effet
	$\mid$	$l = e;$	affectation
	$\mid$	$l =$ `alloc`$(\kappa, n);$	allocation
	$\mid$	`free`$(l);$	déallocation
	$\mid$	`if` (e) `then` $i;$ `else` $i;$	conditionnelle
	$\mid$	`while` (e) $i;$	boucle
	$\mid$	$i\ i$	séquence
	$\mid$	$\{i\}$	bloc d'instructions
	$\mid$	`/*@ assert` $p;$ `*/` i	assertion
Prédicats	$p ::=$	$t \equiv t \mid t \leq t$	comparateurs
	$\mid$	$p \wedge p \mid p \vee p \mid \neg p$	connecteurs
	$\mid$	$\forall (x : \tau);\ t \leq x < t \implies p$	quantificateur universel
	$\mid$	`\valid`(a)	validité d'un pointeur a
	$\mid$	`\initialized`(a)	initialisation de la valeur pointée par a
Termes	$t ::=$	z	entier mathématique
	$\mid$	a	valeur mémoire
	$\mid$	t `+` $t \mid t$ `-` $t \mid t$ `*` $t \mid t$ `/` t	opérateurs mathématiques
	$\mid$	$(\tau)t$	coercion
Types machine	$\kappa ::=$	ι	type entier machine
	$\mid$	$\kappa\star$	type pointeur
Types logiques	$\tau ::=$	κ	type machine
	$\mid$	$\mathbb{Z}$	type des entiers mathématiques

FIGURE 1 – Syntaxe formelle du langage d'étude.

et d'accéder à la valeur pointée ($\star$). L'arithmétique de pointeurs est représentée *via* l'opération a `++` n qui représente le décalage du pointeur a de n fois la taille du type pointé par a (en supposant cette taille fixée pour tout type machine κ).

Les instructions peuvent être des affectations, une allocation dynamique d'une taille correspondant au type en argument multipliée par le nombre d'éléments indiqué, une déallocation, des conditionnelles (la version sans branche `else` peut être simulée grâce à l'instruction sans effet « ; »), des boucles `while`, des séquences ou des blocs d'instructions. Elles peuvent être annotées par des assertions logiques. Ces dernières sont des prédicats permettant de comparer des termes et éventuellement composés de conjonctions et de disjonctions. Ils peuvent aussi être niés et universellement quantifiés. Dans ce dernier cas, la variable quantifiée est un entier syntaxiquement contraint à un intervalle pour garantir qu'une traduction en code puisse être exécutée en temps fini (quoique potentiellement long). Le quantificateur existentiel peut être simulé grâce à la négation. Deux prédicats prédéfinis `\valid` et `\initialized` permettent de tester, respectivement, la validité d'un pointeur et l'initialisation d'une valeur pointée par un pointeur. Les termes sont constitués des expressions entières du langage de programmation, étendues aux entiers mathématiques en précision arbitraire et à un opérateur de coercion (*cast*).

Les noms de chaque variable du programme sont supposés différents deux à deux (aussi bien dans le code que dans les annotations) dans la suite de notre étude, ce qui est toujours possible modulo alpha-conversion. Les programmes sont typés, y compris les annotations logiques. On suppose notamment l'existence d'une famille $\mathcal{Z}$ de types d'entiers machines ι qui contiennent tous au moins les nombres 0 et 1. $\mathcal{Z}$ contient au moins le type `int`. Le système de types sous-jacent est omis par soucis de concision, mais on suppose par la suite que tous les programmes sont bien typés [2]. En particulier, les opérateurs mathématiques dans les annotations sont dans $\mathbb{Z}$. On note $\Sigma(t)$ le type du terme t, tandis que min_ι (resp. max_ι) dénote le plus petit (resp. le plus grand) entier représentable dans le type $\iota \in \mathcal{Z}$. Aussi, $\mathcal{Z}$ est partiellement ordonné par une relation d'ordre $\preccurlyeq$ telle que $\iota_1 \preccurlyeq \iota_2$ si et seulement si tous les entiers représentables dans ι_1 peuvent aussi être représentés dans ι_2. Cette relation est prolongée sur les types logiques entiers en considérant que $\iota \preccurlyeq \mathbb{Z}$ pour tout $\iota \in \mathcal{Z}$.

La sémantique opérationnelle du langage est standard et omise. Néanmoins, il est utile d'en préciser quelques points, en particulier sur le langage d'annotations. D'abord, les prédicats sont à valeur dans $\{0, 1\}$ et une assertion est valide si et seulement si la valeur du prédicat vaut 1 [3]. Ensuite, la sémantique est bloquante [3, 9, 10] : si une erreur survient au cours de l'exécution du programme ou qu'une assertion est invalide, l'exécution du programme est arrêtée. Enfin, on suppose que l'évaluation de chaque terme est bien définie et ne provoque donc aucune erreur à l'exécution. Ainsi, par exemple, aucun terme n'effectue une division par zéro, ne déréférence un pointeur invalide ou n'effectue une coercion qui engendre un débordement. Cette hypothèse peut sembler forte, mais elle est d'une part cohérente avec la sémantique du langage *E-ACSL* [18] et, d'autre part, elle est effectivement garantie par le greffon *E-ACSL* qui génère et compile à la volée les annotations manquantes la garantissant [6].

La définition formelle de l'analyse flot de données présentée dans la section 4 repose sur le graphe de flot de contrôle associé à un programme P. Un tel graphe peut être défini par un sextuplet $(\mathcal{V}, \mathcal{T}, \mathcal{E}, \pi, \mathcal{I}, \mathcal{F})$ [15] dans lequel :

- $\mathcal{V}$ est l'ensemble des sommets du graphe : deux de ces sommets sont associés à chaque instruction du programme P. Ils représentent ses états antérieur et postérieur.
- $\mathcal{T} \subseteq \mathcal{V} \times \mathcal{V}$ est l'ensemble des arcs.
- $\mathcal{E}$ est une fonction des arcs vers l'ensemble des étiquettes $\mathcal{L}$ = {`skip`, $l = e$, $l =$ `alloc`(κ, n), `free`(l), $?e$, $?p$, `erreur`}. La première étiquette représente l'exécution de l'instruction `;`, les deuxième, troisième et quatrième l'exécution respective de l'affectation $l = e$, de l'allocation $l =$ `alloc`(κ, n) et de la déallocation `free`(l), la cinquième un branchement lorsqu'une expression e est vraie, la sixième la poursuite de l'exécution lorsqu'un prédicat p est

2. Cette hypothèse est vérifiée en pratique dans *Frama-C* car l'Arbre de Syntaxe Abstraite (AST) manipulé par les différents greffons est préalablement typé par le noyau de la plateforme.

3. Le choix de valuer les prédicats dans $\{0, 1\}$ plutôt que dans *Prop* facilite l'expression du théorème 4 de la section 3.

valide et la dernière correspond à l'apparition d'une erreur.
— $\mathcal{I} \in \mathcal{V}$ (resp. $\mathcal{F} \in \mathcal{V}$) représente l'état initial (resp. final) du programme P, tandis que $\pi \in \mathcal{V}$ représente son état d'erreur.

Le détail de la construction du graphe est omis, mais est standard [15]. Signalons tout de même qu'une instruction `/*@ assert p; */` i est décomposée en l'état avant l'assertion, l'état intermédiaire une fois l'assertion vérifiée, mais avant l'exécution de i et le ou les état(s) relatif(s) à i (plus l'état d'erreur atteignable si p est invalide ou si l'exécution de i provoque une erreur).

3. Système de types pour pouvoir utiliser des entiers machines

Problématique Le système de types présenté ici a pour but de diminuer au maximum le nombre d'instructions *GMP* généré par le traducteur d'*E-ACSL*. Supposons dorénavant dans nos exemples que $\mathcal{Z}$ est limité aux types `int` et `char` et que nous disposons d'une architecture matérielle – certes assez limitée et peu réaliste – de 8 bits, dans laquelle les valeurs de type `int` seraient comprises entre -128 et 127, tandis que celles du type `char` seraient comprises entre -32 et 31. On a donc `char` $\preccurlyeq$ `int`. Considérons à présent les deux assertions suivantes dans lesquelles x est une variable de type `int`, tandis que c est une variable de type `char`.

`/*@ assert` $x + 1 \leq 127$; `*/`
`/*@ assert` $c + 1 \equiv 0$; `*/`

L'addition présente dans ces assertions correspond à l'addition mathématique en précision arbitraire. Par exemple, la première assertion n'est pas vérifiée pour $x = 127$. Il n'est pas toujours possible de traduire de manière correcte les assertions directement vers les expressions C correspondantes. Ici, la traduction directe de la première assertion en une assertion C de la forme `__e_acsl_assert(x + 1 <= 127);` serait erronée, car ajouter 1 à x de type `int` provoquerait un débordement arithmétique pour $x = 127$, donc la traduction en C ne pourrait pas échouer. Ainsi le greffon *E-ACSL* génère du code utilisant des entiers en précision arbitraire et les opérations mathématiques correspondantes *via* la bibliothèque *GMP*. Par exemple, la figure 2 montre la façon (très légèrement simplifiée[4]) dont *E-ACSL* traduit notre exemple en *C*. En *GMP*, il est en effet nécessaire de déclarer, d'allouer et, le cas échéant, d'initialiser un pointeur par entier (fonctions `__gmpz_init`$*$) et d'effectuer un appel de fonction par opération (ici une addition avec `__gmpz_add` et une comparaison avec `__gmpz_cmp`). Comme de la mémoire est allouée, il faut finalement la libérer *via* des appels à `__gmpz_clear`.

Ce schéma de traduction est plus complexe et plus coûteux en temps d'exécution et en mémoire que la traduction qui serait naturelle (mais potentiellement incorrecte) en l'instruction `__e_acsl_assert(x + 1 <= 127);`. Néanmoins, sans information supplémentaire, le même type de traduction serait effectuée pour la seconde assertion `/*@ assert c + 1 == 0; */`. Pourtant, avec notre architecture hypothétique, il est tout à fait possible de vérifier cette assertion en utilisant des entiers machines. En effet, les constantes 0 et 1 sont toutes deux représentables en machine, alors que $c + 1$ est une expression dont la valeur est comprise entre -31 et 32 et ne peut donc pas déborder dans le type `int` (elle le peut néanmoins dans le type `char`). Il serait donc tout à fait correct de générer directement l'instruction `__e_acsl_assert(c + 1 == 0);`. Le but de notre système de types est de distinguer ce cas dans lequel on peut utiliser l'arithmétique machine, du précédant dans lequel l'utilisation de *GMP* est requise. Une contrainte supplémentaire est la rapidité : *E-ACSL* doit générer le programme instrumenté rapidement, à la manière d'un compilateur.

Inférence d'intervalles Ce système de types repose sur une inférence d'intervalles qui détermine, pour chaque terme d'un type entier, une sur-approximation correcte de ses valeurs possibles. Cette

4. Ici nous avons omis des coercions, ainsi que des arguments supplémentaires de la fonction `__e_acsl_assert` qui permettent de produire des messages d'erreur plus précis.

```
/*@ assert x + 1 <= 127; */
{
  /* déclaration des variables GMP temporaires */
  mpz_t __e_acsl_x;
  mpz_t __e_acsl;
  mpz_t __e_acsl_add;
  mpz_t __e_acsl_2;
  int __e_acsl_le;
  /* calcul de x+1 */
  __gmpz_init_set_si(__e_acsl_x, x);
  __gmpz_init_set_si(__e_acsl, 1);
  __gmpz_init(__e_acsl_add);
  __gmpz_add(__e_acsl_add, __e_acsl_x, __e_acsl);
  /* comparaison de x+1 et 127 */
  __gmpz_init_set_si(__e_acsl_2, 127);
  __e_acsl_le = __gmpz_cmp(__e_acsl_add, __e_acsl_2);
  e_acsl_assert(__e_acsl_le <= 0);
  /* libération de la mémoire */
  __gmpz_clear(__e_acsl_x);
  __gmpz_clear(__e_acsl);
  __gmpz_clear(__e_acsl_add);
  __gmpz_clear(__e_acsl_2);
}
```

FIGURE 2 – Exemple de code généré par *E-ACSL* et utilisant *GMP*.

inférence a été introduite brièvement dans un article précédant [6]. Elle est ici présentée figure 3. On note $[a;b]$ l'intervalle des valeurs comprises entre a et b, avec $a \leq b$ et chacune des bornes incluse. Par plongement dans les ensembles, nous utilisons ici les opérateurs ensemblistes habituels, en particulier l'union $\cup$ et l'intersection $\cap$. Par ailleurs, le système d'inférence repose sur un environnement de typage Γ qui associe un intervalle à chaque variable logique liée à un quantificateur universel.

$$\frac{}{\Gamma \vDash z : [z;z]} \qquad \frac{x \in \Gamma}{\Gamma \vDash x : \Gamma(x)} \qquad \frac{x \notin \Gamma \quad \Sigma(x) = \iota}{\Gamma \vDash x : [min_\iota; max_\iota]} \qquad \frac{\Sigma(a) = \iota\star}{\Gamma \vDash \star a : [min_\iota; max_\iota]}$$

$$\frac{\Gamma \vDash t_1 : [l_1;u_1] \quad \Gamma \vDash t_2 : [l_2;u_2] \quad \texttt{op} \in \{\texttt{+};\texttt{-};\texttt{*};\texttt{/}\} \quad (\texttt{op} \neq \texttt{/} \text{ ou } 0 \notin [l_2,u_2])}{\Gamma \vDash t_1 \ \texttt{op} \ t_2 : [\min(l_1 \ op \ l_2, l_1 \ op \ u_2, u_1 \ op \ l_2, u_1 \ op \ u_2); \max(l_1 \ op \ l_2, l_1 \ op \ u_2, u_1 \ op \ l_2, u_1 \ op \ u_2)]}$$

$$\frac{\Gamma \vDash t_1 : [l_1;u_1] \quad \Gamma \vDash t_2 : [l_2;u_2] \quad l_2 < 0 < u_2}{\Gamma \vDash t_1 / \ t_2 : [\min(l_1, -u_1); \max(-l_1, u_1)]}$$

$$\frac{\Gamma \vDash t : I}{\Gamma \vDash (\iota)t : I \cap I_\iota}$$

FIGURE 3 – Inférence d'intervalles.

À chaque constante, l'intervalle singleton correspondant est inféré. L'intervalle associé à une variable logique est fourni par l'environnement (l'ajout de la variable dans l'environnement est effectué par l'introduction du quantificateur universel dans le système de types qui est présenté figure 4 et qui est l'unique utilisateur de cette inférence d'intervalles). L'intervalle associé à une variable x du programme est celui contenant toutes les valeurs du type de x. Cette approximation est correcte et peu coûteuse à calculer, mais relativement grossière. Il est en réalité possible d'utiliser ici n'importe quelle analyse de valeurs de programmes comme celle proposée par le greffon *Value* de *Frama-C* [5, 12]. De même, l'intervalle inféré lors d'un accès à un entier *via* un pointeur est l'intervalle contenant toutes les

valeurs du type pointé. Pour les opérateurs, n'importe quelle arithmétique d'intervalles correcte [17] peut être utilisée (celle indiquée ici est la plus précise possible, quoique peu efficace à calculer). Une difficulté existe dans le cas d'une division pour laquelle l'intervalle du dénominateur contiendrait 0. Dans la figure 3, nous traitons séparément (sans rechercher la forme optimale) le cas où l'intervalle du dénominateur ne contient pas 0, et le cas où il contient 0 strictement à l'intérieur. Comme *E-ACSL* garantit par ailleurs qu'une division par zéro n'est pas possible (cf. section 2), nous pouvons en effet considérer qu'une éventuelle borne inférieure (resp. supérieure) nulle est remplacée par 1 (resp. -1). Enfin, dans le cas d'une coercion $(\iota)t$, l'intervalle inféré est l'intersection entre toutes les valeurs possibles pour le type ι et toutes celles possibles pour le terme t. Là encore, *E-ACSL* garantit que cette coercion est sûre. Enfin, aucune règle n'existe pour la prise d'adresse $\&l$, ni pour le décalage de pointeur a `++` n : ces termes ne sont pas des entiers.

Les deux théorèmes suivants (prouvés par induction simple sur la structure des termes) garantissent la correction et une relative précision de l'inférence d'intervalles. Ils reposent sur des notions de cohérence entre environnement d'intervalles Γ, environnement de typage Σ et environnement d'évaluation Λ. Cet environnement d'évaluation, dont la définition formelle est omise, associe à chaque variable une valeur (ici, un entier ou une adresse mémoire). Ainsi, Λ est dit cohérent avec Γ si toutes les variables logiques de son domaine définissent le domaine de Γ et que la valeur associée à chacune [5] est dans l'intervalle associé dans Γ. De manière similaire, Λ est cohérent avec Σ si toutes les variables du programme de son domaine sont dans le domaine de Σ et que la valeur associée à chacune peut être du type associé dans Σ. L'environnement de typage Σ utilisé dans le système de règle est toujours supposé cohérent avec un environnement d'évaluation Λ donné.

Théorème 1 (Correction de l'inférence d'intervalles) *Étant donné un environnement d'inférence Γ et un environnement d'évaluation Λ cohérents, l'ensemble des valeurs vers lesquelles un terme t peut être évalué dans Λ est inclus dans l'intervalle inféré pour t dans Γ.*

Théorème 2 (Précision relative de l'inférence d'intervalles) *Étant donné un environnement d'inférence Γ et un environnement de typage Σ cohérents, l'intervalle inféré pour t dans Γ est inclus dans l'ensemble des valeurs représentées par $\Sigma(t)$.*

Système de types Maintenant que nous pouvons inférer un intervalle raisonnablement précis pour chaque terme d'un type entier, nous pouvons définir un système de types dont le but est d'obtenir deux informations : le type du résultat et celui de l'opération éventuelle correspondant à l'expression à générer à partir d'un terme. Ces deux types ne sont pas forcément les mêmes. Par exemple, si z est une constante entière non représentable, le prédicat $z \equiv 0$ doit être traduit en utilisant une comparaison *GMP* (le type associé est alors $\mathbb{Z}$), mais le résultat, valant soit 0 soit 1, peut être de type `int`. Pour mieux comprendre le rôle du système de types, considérons un exemple un peu plus complexe : $1+(x+1)/(y-64)$ avec x de type `int` et y de type `char`. Ici, comme $x+1$ peut déborder, l'opération doit être effectuer à l'aide de *GMP*. Il faut donc aussi traduire x et 1 en *GMP* avant d'effectuer l'opération. Par contre l'opération $y - 64$ peut être effectuée dans le type `int` et il suffit de coercer y de `char` vers `int` (ce qui est inutile en pratique, une coercion implicite étant automatiquement introduite). L'analyse d'intervalles permet de garantir que le résultat de cette soustraction est un nombre entre -96 et -33. Par suite, elle permet aussi de garantir que le résultat de la division est entre -3 et 3. Comme une des opérandes de la division est un *GMP*, elle doit néanmoins être effectuée en *GMP*. Cependant, son résultat (appelons-le z) peut, de manière sûre, être converti en `int` de manière à calculer $1 + z$ avec des entiers machines.

Plus généralement, le type de l'opération doit être suffisamment grand pour contenir la valeur de son résultat et celles de ses opérandes. Par conséquent, le jugement de typage des termes est $\Gamma \vdash t : \tau_1 \rightsquigarrow \tau_2$ et signifie « dans l'environnement Γ, le résultat de la traduction de t peut être de

5. On rappelle que, dans notre étude, toutes les variables logiques sont des entiers.

type τ_1 et l'opération doit être effectuée avec le type τ_2 ». En particulier, cette définition induit que τ_1 n'est pas forcément unique : notre système est fondé sur du sous-typage. Lorsqu'aucune opération n'est associée (par exemple pour les constantes) ou lorsque τ_2 n'est pas utile, on peut simplifier le jugement en $\Gamma \vdash t : \tau_1$. Ce jugement est étendu aux prédicats et est alors noté $\Gamma \vdash_p p : \tau_1 \rightsquigarrow \tau_2$, avec potentiellement la même simplification pour omettre τ_2. Par ailleurs, on note $\theta(I)$ le type $\preccurlyeq$-maximum entre `int` et le plus petit type contenant I. L'idée sous-jacente est de calculer le type le plus précis possible pour représenter I, mais qu'il est contre-productif d'avoir une précision supérieure à `int` : en C, les constantes sont au moins de types `int` et, d'après les règles de typage du C, les opérations les manipulant aussi. Être trop précis ne servirait qu'à introduire des coercions inutiles vers `int` (par exemple de `char` vers `int`). Le système de types est présenté figure 4.

$$\frac{\Gamma \vDash z : I}{\Gamma \vdash z : \theta(I)} \qquad \frac{\Gamma \vDash a : I}{\Gamma \vdash a : \theta(I)} \qquad \frac{\Gamma \vDash (\iota)t : I \quad \Gamma \vDash t : I_t \quad \tau = \theta(I \cup I_t) \quad \Gamma \vdash t : \tau}{\Gamma \vdash (\iota)t : \tau \rightsquigarrow \tau}$$

$$\frac{\Gamma \vDash t_1 : I_1 \quad \Gamma \vDash t_2 : I_2 \quad \Gamma \vDash t_1 \ \texttt{op}\ t_2 : I \quad \tau = \theta(I_1 \cup I_2 \cup I) \quad \Gamma \vdash t_1 : \tau \quad \Gamma \vdash t_2 : \tau \quad \texttt{op} \in \{\texttt{+};\texttt{-};\texttt{*};\texttt{/}\}}{\Gamma \vdash_p t_1 \ \texttt{op}\ t_2 : \tau \rightsquigarrow \tau}$$

$$\frac{\Gamma \vdash t : \tau' \quad \tau' \preccurlyeq \tau}{\Gamma \vdash t : \tau} \qquad \frac{\Gamma \vdash t : \tau' \rightsquigarrow \tau' \quad \tau' \succ \tau \quad \Gamma \vDash t : I \quad \theta(I) \preccurlyeq \tau}{\Gamma \vdash t : \tau \rightsquigarrow \tau'}$$

$$\frac{\Gamma \vDash t_1 : I_1 \quad \Gamma \vDash t_2 : I_2 \quad \tau = \theta(I_1 \cup I_2) \quad \Gamma \vdash t_1 : \tau \quad \Gamma \vdash t_2 : \tau \quad op \in \{\equiv; \leq\}}{\Gamma \vdash_p t_1 \ op\ t_2 : \texttt{int} \rightsquigarrow \tau}$$

$$\frac{\Gamma \vdash_p p_1 : \texttt{int} \quad \Gamma \vdash_p p_2 : \texttt{int}}{\Gamma \vdash_p p_1 \wedge p_2 : \texttt{int}} \qquad \frac{\Gamma \vdash_p p_1 : \texttt{int} \quad \Gamma \vdash_p p_2 : \texttt{int}}{\Gamma \vdash_p p_1 \vee p_2 : \texttt{int}} \qquad \frac{\Gamma \vdash_p p : \texttt{int}}{\Gamma \vdash_p \neg p : \texttt{int}}$$

$$\frac{\begin{array}{c}\Gamma \vDash t_1 : [l_1; u_1] \quad \Gamma \vDash t_2 : [l_2; u_2] \quad \tau' = \theta([l_1; u_1], [l_2; u_2]) \\ \Gamma \vdash t_1 : \tau' \quad \Gamma \vdash t_2 : \tau' \quad \Gamma, x : ([l_1; u_2] \cap I_\tau) \vdash_p p : \texttt{int}\end{array}}{\Gamma \vdash_p \forall (x : \tau);\ t_1 \leq x < t_2 \implies p : \texttt{int}}$$

$$\frac{}{\Gamma \vdash_p \backslash\texttt{valid}(a) : \texttt{int}} \qquad \frac{}{\Gamma \vdash_p \backslash\texttt{initialized}(a) : \texttt{int}}$$

FIGURE 4 – Système de types.

Seuls les termes d'un type entier ont besoin d'être typés. Les cas de base sont les constantes et les valeurs mémoires pour lesquels le type le plus précis est celui déterminé par l'inférence d'intervalles. Le type τ le plus précis d'une opération arithmétique est le plus petit type permettant de représenter à la fois les opérandes et le résultat. L'opération doit aussi être effectuée dans τ. Par exemple, avec notre architecture 8-bits, le type le plus précis de 120 + 8 est $\mathbb{Z}$ car le calcul peut déborder et devra donc être fait à l'aide de *GMP*, tandis que celui de 127 * (−1) + 1 est `int`. En tenant compte de la monotonie des opérateurs mathématiques, il n'est en réalité pas toujours nécessaire d'inférer les intervalles à la fois des opérandes et du résultat. Par exemple, une addition est croissante envers chacun de ses opérandes : leurs intervalles sont donc inutiles car toujours inclus dans l'intervalle du résultat. Ces optimisations sont omises ici. Le cas de la coercion est similaire à celui d'un opérateur.

Les deux règles supplémentaires sur les termes sont les règles de sous-typage de notre système. La première est la règle de subsomption [16] : elle spécifie qu'on peut utiliser un sous-type à la place du type attendu. Ici par exemple, lorsqu'un *GMP* est attendu, un `int` peut aussi être utilisé car $\texttt{int} \preccurlyeq \mathbb{Z}$. Lors de la traduction effective par *E-ACSL*, utiliser cette règle pour obtenir le type $\mathbb{Z}$ indique qu'il est nécessaire de convertir un entier machine en *GMP* comme lors du calcul 120 + 8 pour lequel les deux constantes en argument doivent l'être. La règle suivante est duale et permet de convertir par exemple un *GMP* en `int` lorsque cela est correct, c'est-à-dire lorsque l'intervalle résultant est inclus dans `int`. Par exemple, calculer 128 / 2 requiert l'utilisation de *GMP* (à cause de 128) mais le résultat

peut, de manière sûre, être converti en `int` en fonction du contexte (par exemple pour le comparer à 0). En général, un inconvénient d'utiliser de tels règles de sous-typage est que le système ainsi écrit ne définit pas un algorithme car plusieurs règles sont potentiellement applicables. Néanmoins, dans notre cas particulier, il est facile de surmonter cette difficulté en n'utilisant les règles de sous-typage que lorsqu'aucun autre choix est possible. Cette stratégie permet aussi de n'introduire qu'un nombre minimal de coercions (en particulier de ou vers *GMP*). Un arbre de dérivation pour typer l'exemple précédent $1 + (x + 1)/(y - 64)$ dans un environnement $\Gamma = x : [-128; 127], y : [-32; 31]$ est donné figure 5. L'inférence d'intervalles y est omise mais a déjà été expliquée informellement pour cet exemple. Les étapes marquées $\preccurlyeq$ correspondent à l'utilisation d'une subsomption. On y remarque aussi la conversion du résultat *GMP* de la division en `int`.

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma \vdash x : \texttt{int}}{\Gamma \vdash x : \mathbb{Z}} \preccurlyeq \dfrac{\Gamma \vdash 1 : \texttt{int}}{\Gamma \vdash 1 : \mathbb{Z}}}{\Gamma \vdash x + 1 : \mathbb{Z}} \preccurlyeq \dfrac{\dfrac{\Gamma \vdash y : \texttt{int} \qquad \Gamma \vdash 64 : \texttt{int}}{\Gamma \vdash y - 64 : \texttt{int}}}{\Gamma \vdash y - 64 : \mathbb{Z}}}{\Gamma \vdash (x + 1)/(y - 64) : \mathbb{Z} \rightsquigarrow \mathbb{Z}} \preccurlyeq \qquad [-3; 3] \preccurlyeq \texttt{int}}{\Gamma \vdash 1 : \texttt{int} \qquad \Gamma \vdash (x + 1)/(y - 64) : \texttt{int} \rightsquigarrow \mathbb{Z}}}{\Gamma \vdash 1 + (x + 1)/(y - 64) : \texttt{int}}$$

FIGURE 5 – Exemple de dérivation.

Typer les prédicats ne pose pas de difficultés particulières : le résultat de leur traduction est soit 0 soit 1, et le type le plus précis est donc `int`. La règle des opérateurs de comparaison est similaire à celle des opérateurs arithmétiques mais exploite directement le fait que l'intervalle résultant est $[0; 1]$ et n'a donc aucun impact sur le type (tous les types entiers contiennent ces valeurs). Aussi, la règle de la quantification universelle introduit la variable liée dans l'environnement de typage. Un détail important est le fait que la borne supérieure t_2 de x est syntaxiquement stricte alors que l'intervalle associé à x dans Γ contient la borne supérieure u_2 de t_2. Ce n'est pas une sur-approximation inutile. Au contraire, c'est nécessaire à la correction. En effet, une quantification universelle est traduite par *E-ACSL* en une boucle `for` (équivalente à un `while`) dont l'indice varie de t_1 (inclus) à t_2 (exclu). Néanmoins, de manière classique, une incrémentation supplémentaire est effectuée juste avant de sortir de la boucle, ce que reflète le système de types. En particulier, si t_2 est le successeur du plus grand entier représentable, ne pas utiliser la traduction vers *GMP* (ce qui serait le cas sans l'écart d'une unité) engendrerait un débordement qui aurait pour conséquence de rendre le test de sortie de boucle indéfiniment faux : le programme généré ne terminerait pas.

Les deux théorèmes suivants garantissent la correction du système de types, aussi bien pour le type calculé que pour celui éventuellement associé à l'opération effectuée. La preuve du premier est une induction sur les règles de sémantique opérationnelle et s'appuie sur les théorèmes 1 et 2, tandis que le second est un corollaire des trois autres théorèmes.

Théorème 3 (Correction du système de types) *Étant donné un environnement d'inférence* Γ *et un environnement d'évaluation* Λ *cohérents, si* t *est de type* τ *dans* Γ *d'après le système de la figure 4, alors l'ensemble des valeurs vers lesquelles un terme* t *peut être évalué dans* Λ *est inclus dans les valeurs représentées par* τ*.*

Théorème 4 (Correction du système des opérations) *Étant donné un environnement d'inférence* Γ *et un environnement d'évaluation* Λ *cohérents, si l'opération associée à un terme* t *(resp. predicat* p*) est de type* τ *dans* Γ *d'après le système de la figure 4 (c'est-à-dire* $\Gamma \vdash t : \_ \rightsquigarrow \tau$ *ou, resp.,* $\Gamma \vdash_p p : \_ \rightsquigarrow \tau$*), alors l'ensemble des valeurs vers lesquelles* t *(resp.* p*) et ses opérandes peuvent être chacun évalués dans* Λ *est inclus dans les valeurs représentées par* τ*.*

4. Analyse de données pour réduire l'instrumentation

Problématique L'analyse flot de données présentée ici a pour but de diminuer au maximum le nombre d'appels à la bibliothèque modélisant la mémoire. Malgré les efforts déployés pour implanter cette bibliothèque de manière efficace [13], une telle instrumentation est très invasive et coûteuse aussi bien en temps de calcul qu'en consommation mémoire. Ainsi, la figure 6 montre l'instrumentation effectuée pour un petit programme *C* dans lequel la validité d'un pointeur est vérifiée[6]. Les lignes générées par l'instrumentation sont sur-indentées par rapport aux lignes du programme initial : les lignes de code du programme instrumenté sont ici 170% plus nombreuses que celles du programme initial et, en outre, les lignes ajoutées sont de coûteux appels de fonctions.

```
int main(void) {
  int x, y, z, *p;
      /* allocation des variables locales */
      __store_block((void *)(& p), 4U);
      __store_block((void *)(& z), 4U);                              /* superflu */
      __store_block((void *)(& y), 4U);                              /* superflu */
      __store_block((void *)(& x), 4U);
      __full_init((void *)(& p));  /* initialisation de p */
  p = &x;
      __full_init((void *)(& x));  /* initialisation de x */         /* superflu */
  x = 0;
      __full_init((void *)(& y));  /* initialisation de y */         /* superflu */
  y = 1;
      __full_init((void *)(& z));  /* initialisation de z */         /* superflu */
  z = 2;
  /*@ assert \valid(p); */
      /* test de validité */
      {
        int __e_acsl_initialized;
        int __e_acsl_and;
        __e_acsl_initialized = __initialized((void *)(& p),sizeof(int *));
        if (__e_acsl_initialized) {
          int __e_acsl_valid;
          __e_acsl_valid = __valid((void *)p, sizeof(int));
          __e_acsl_and = __e_acsl_valid;
        }
        else __e_acsl_and = 0;
        e_acsl_assert(__e_acsl_and);
      }
  *p = 3;
      /* libération de la mémoire */
      __delete_block((void *)(& p));
      __delete_block((void *)(& z));                                 /* superflu */
      __delete_block((void *)(& y));                                 /* superflu */
      __delete_block((void *)(& x));
  return 0;
}
```

FIGURE 6 – Exemple d'instrumentation mémoire complète effectuée par *E-ACSL*.

6. Cet exemple suppose une architecture 32 bits.

En effet, toutes les opérations sur la mémoire sont instrumentées : les allocations (qui sont ici induites par les déclarations de variables) *via* un appel à la fonction `__store_block`, les initialisations *via* un appel à la fonction `__full_init` et les libérations (qui sont ici induites par la sortie du block principal de la fonction) *via* un appel à la fonction `__delete_block`. À ces lignes, il faut ajouter celles permettant de tester le prédicat de validité, *via* notamment les appels de fonctions `__initialized` et `__valid` (avant de lire un pointeur p pour vérifier sa validité, il faut d'abord vérifier qu'il est correctement initialisé car, sinon, sa lecture provoque un comportement indéfini et est donc interdite).

Cependant, ce programme ne vérifie que la validité du pointeur p. Ainsi, alors qu'instrumenter $\&x$ est requis car il s'agit d'un alias de p, instrumenter y et z ainsi que l'initialisation de x est superflu et il conviendrait de ne pas générer les instructions correspondantes marquées `/* superflu */`.

Analyse flot de données Cette optimisation de l'instrumentation est effectuée par *E-ACSL* grâce à une analyse flot de données arrière sur-approximante paramétrée par une analyse d'alias. On suppose ici l'existence d'une fonction $\mathcal{A}$, correspondant à une analyse d'alias, et prenant en argument une adresse a et retournant une sur-approximation de l'ensemble des adresses en alias avec a, en incluant a elle-même. Pour éviter toute confusion pour un pointeur p entre adresse pointée $\star p$ et valeur gauche p, nous allons parler d'alias et d'observation des *adresses,* donc, observer l'adresse $\&l$ pour une valeur gauche l. Pour les tableaux, nous observons les adresses de base plutôt que chaque élément séparément (donc tous les éléments d'un tableau seront observés dès qu'un élément nécessite l'observation). La fonction $\mathcal{A}$ est également supposée traiter les éléments à décalage près : un pointeur en alias avec un élément d'un tableau est considéré en alias avec la base du tableau.

Comme pour le système de types, dans le contexte de *E-ACSL*, la rapidité est importante, plus importante même que la précision. Cette analyse peut donc être par exemple celle de Steensgard [19] ou, lorsqu'elle a été pré-calculée, l'analyse de valeurs de *Frama-C* [5, 12]. Pour ne pas alourdir les notations introduites par la suite, la fonction $\mathcal{A}$ est supposée ici insensible au contexte (*context-insensitive*, comme celle de Steensgard [19]), mais une analyse sensible au contexte (*context-sensitive*, comme celle du greffon *Value* de *Frama-C* [5]) s'utilise de la même façon. Munie de cette fonction, notre analyse calcule, en chaque point de programme, une sur-approximation de l'ensemble des valeurs gauches et des adresses utilisées ultérieurement dans une annotation.

Pour définir notre analyse formellement, nous nous plaçons dans un cadre d'analyse similaire à celui de Nielson *et al.* [15] en la définissant comme un quadruplet $(\mathcal{S}, \subseteq, \epsilon, s_0)$ défini ci-après. Ce cadre permet de déterminer automatiquement l'état obtenu en chaque point de programme, y compris en présence de boucle.

— $(\mathcal{S}, \subseteq)$ est un treillis d'ensembles de couples (a, r), où a est une adresse et $r \in \{i, v\}$ une raison d'observation, ordonnés par inclusion et représentant l'état de l'analyse. Cet ensemble représente les adresses à observer par instrumentation et la raison de leur instrumentation (i pour « initialisation » et v pour « validité »).

— $s_0 \in \mathcal{S}$ est l'état initial de l'analyse, c'est-à-dire – dans le cas d'une analyse arrière telle que la notre – l'état correspondant au point de sortie du programme : aucun accès mémoire n'a besoin d'être instrumenté à la fin du programme et, donc, $s_0 = \emptyset$.

— $\epsilon : \mathcal{L} \to \mathcal{S} \to \mathcal{S}$ est un ensemble de fonctions de transition, monotones sur les états et dépendantes des étiquettes portées par les arêtes du graphe de flot de contrôle. Il est détaillé figure 7 et, outre l'opérateur $\mathcal{A}$, il dépend d'un opérateur *base* présenté dans la même figure et calculant l'adresse de base correspondant à une valeur mémoire de type pointeur : le but de cet opérateur est d'oublier les décalages éventuellement effectués. Dans cette figure et par la suite, on note $\mathcal{A}_a^R$ l'ensemble $\{(x, r) \mid x \in \mathcal{A}(\text{base}(a)), r \in R\}$, et simplement $\mathcal{A}_a^r$ pour un singleton $R = \{r\}$.

Les seuls cas de la fonction de transition ϵ qui ne se contentent pas de propager l'état sont les cas du test de validité (resp. d'initialisation) d'un pointeur (resp. d'une valeur pointée), de l'affectation et des allocations et déallocations. Pour le test de validité (resp. d'initialisation), il faut ajouter à

$$
\begin{aligned}
\epsilon_{;}(\sigma) &= \sigma \\
\epsilon_{l=e}(\sigma) &= \begin{cases} \theta_l^\sigma \cup \mathcal{A}_e^R & \text{si } l \text{ est un pointeur, et } R = \{r \mid (\text{base}(l), r) \in \sigma\} \\ \theta_l^\sigma & \text{sinon} \end{cases} \\
\epsilon_{l=\texttt{alloc}(\kappa,n)}(\sigma) &= \theta_l^\sigma \\
\epsilon_{\texttt{free}(l)}(\sigma) &= \sigma \\
\epsilon_{?e}(\sigma) &= \sigma \\
\epsilon_{?\backslash\texttt{valid}(a)}(\sigma) &= \sigma \cup \mathcal{A}_a^v \cup \mathcal{A}_a^i \\
\epsilon_{?\backslash\texttt{initialized}(a)}(\sigma) &= \sigma \cup \mathcal{A}_a^i \\
\epsilon_{?(p_1 \wedge p_2)}(\sigma) &= \epsilon_{?p_1}(\sigma) \cup \epsilon_{?p_2}(\sigma) \\
\epsilon_{?(p_1 \vee p_2)}(\sigma) &= \epsilon_{?p_1}(\sigma) \cup \epsilon_{?p_2}(\sigma) \\
\epsilon_{?\neg p}(\sigma) &= \epsilon_{?p}(\sigma) \\
\epsilon_{?\forall\,(x:\tau);\, t_1 \le x \le t_2 \implies p}(\sigma) &= \epsilon_{?p}(\sigma) \\
\epsilon_{?(t_1\, op\, t_2)}(\sigma) &= \sigma \qquad \text{où } op \in \{\equiv, \le\} \\
\theta_l^\sigma &= \begin{cases} \sigma \setminus \{(\&l, i)\} & \text{si } l = x \text{ est une variable,} \\ \sigma & \text{sinon} \end{cases}
\end{aligned}
$$

$$
\begin{aligned}
\text{base}(x) &= x & \text{base}(\star a) &= \star\,\text{base}(a) \\
\text{base}(a \texttt{ ++ } n) &= \text{base}(a) & \text{base}(\&x) &= \&x \\
\text{base}(\&(\star a)) &= \text{base}(a)
\end{aligned}
$$

FIGURE 7 – Analyse flot de données arrière sur-approximante.

l'état σ le fait de devoir observer le pointeur (resp. la valeur pointée) testé (resp. testée) et tous ses alias potentiels. Comme tester la validité implique de tester l'initialisation, il faut observer ces deux informations. Dans le cas d'une allocation de l, on sait que l vient d'être initialisé et il n'est donc plus nécessaire d'observer cette information si $l = x$ est une variable (si $l = \star a$ peut être une case de tableau on continue à l'observer à cause des autres éléments du tableau, voir la définition de θ_l^σ). Comme l'allocation peut échouer, il n'en est pas en revanche de même pour la validité. Enfin, pour une affectation dont le membre gauche est observé, il n'est plus nécessaire d'observer son initialisation s'il s'agit d'une variable, mais il faut en revanche observer les mêmes informations pour le membre droit et ses alias potentiels s'il s'agit d'un pointeur. De manière traditionnelle dans une analyse de type « variable vivante », il faut effectuer la suppression du membre gauche avant l'ajout du membre droit pour prendre correctement en compte les affectations comme $x = x + 1$; dans lesquelles l'initialisation de x doit continuer à être observée.

5. Implémentation et expérimentations

Le schéma de traduction du greffon *E-ACSL* de *Frama-C* est présenté figure 8. L'implantation actuelle contient notamment à la fois une analyse flot de données arrière interprocédurale et un système de types, tous deux similaires dans leur principe à ceux présentés dans cet article, tout en étant chacun moins précis, car ne bénificiant pas de certaines optimisations présentées ici. Par exemple, si une opération nécessite d'être effectuée en *GMP* mais que son résultat pourrait être représenté par un entier machine, l'implantation actuelle continue les calculs suivants en *GMP* alors que le système présenté ici les continue avec des entiers machines, tandis que, au niveau de l'analyse de

données, la raison d'observation n'est actuellement pas calculée, ce qui induit des sur-approximations supplémentaires et une génération de code sous-optimisée. Coder les versions présentées ici est en cours. Alors que l'analyse flot de données arrière est une analyse effectuée avant l'instrumentation principale, le typage est effectué à la volée, dès qu'un prédicat a besoin d'être traduit.

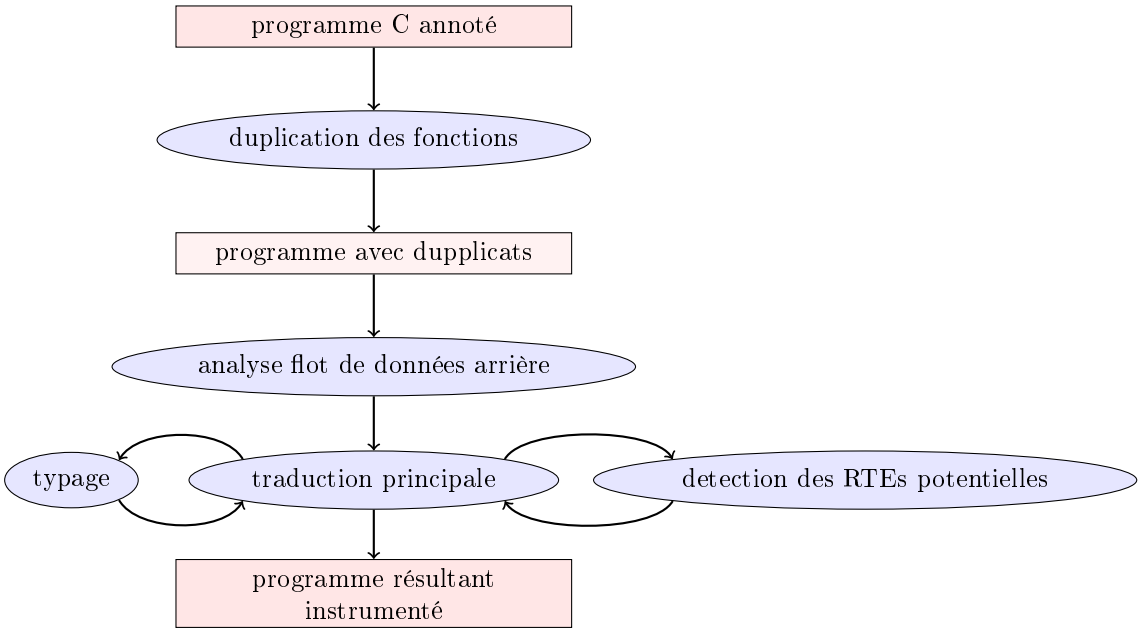

FIGURE 8 – Schéma de traduction du greffon E-ACSL.

Malgré le fait que les analyses actuelles soient moins précises que celles présentées ici, nous pouvons néanmoins mesurer en pratique leur efficacité. Quelques résultats expérimentaux sont présentés figure 9, obtenus sur un Intel Core i7-3520M 2.90GHz avec 16Go de RAM. Il s'agit principalement de programmes effectuant des opérations classiques sur des tableaux (alloués statiquement ou dynamiquement), des arbres ou des matrices contenant plusieurs dizaines de milliers d'éléments. La colonne « base » indique le temps d'exécution du programme non instrumenté, compilé avec *GCC*. Les colonnes suivantes indiquent successivement les temps d'exécution des programmes instrumentés par *E-ACSL* (et compilés de la même manière) sans aucune optimisation, sans le système de types mais avec l'analyse flot de données, sans l'analyse flot de données mais avec le système de types et avec les deux analyses actives en même temps. La dernière colonne fournit à titre de comparaison le temps d'exécution du binaire instrumenté avec *Valgrind* [14], un des outils d'analyse dynamique les plus utilisés pour détecter des violations mémoires. Les pourcentages indiqués entre parenthèses représente le gain de temps d'exécution par rapport à la version non optimisée.

On peut constater que chacune des deux optimisations, considérées séparément, se comporte, au pire, de manière similaire au programme sans optimisation, à deux exceptions notables près (*quickSort* et *insertSort*) pour l'analyse flot de données arrière. Ces exceptions viennent du fait que, en présence de *GMP*, *E-ACSL* ne génère pas, à l'heure actuelle, de vérification pour les accès pointeurs. Par exemple, si, dans une annotation, $*p$, de type *int*, peut être compilé avec des entiers machine, alors une garde est générée pour vérifier dynamiquement que ce déréférencement est valide avant de l'exécuter. Cette garde n'est pas générée si $*p$ doit être converti en *GMP*. Dans les deux exemples considérés, cela aboutit à la génération de plusieurs gardes supplémentaires provoquant de mauvaises performances. Les cas où le typage ne provoque (quasiment) aucun gain de performance correspondent aux cas dans lesquels peu ou pas d'opérations arithmétiques sont effectués : utiliser ou non *GMP* pour ces

Nom du test	base	non optimisé	sans typage	sans flot de données	optimisé	*Valgrind*
bubbleSort	0.58	73.52	18.32 (75%)	51.4 (30%)	4.13 (94%)	10.56
binSearch	0.01	132.17	132.22 (0%)	2.88 (98%)	2.82 (98%)	0.46
mergeSort	0.04	106.02	83.22 (22%)	103.26 (3%)	92.85 (12%)	2.73
quickSort	0.01	3.1	1.39 (55%)	40.21 (-92%)	1.15 (63%)	0.51
RedBlTree	0.02	0.71	0.58 (18%)	0.71 (0%)	0.59 (17%)	2.21
merge	0.01	0.76	0.68 (11%)	0.69 (9%)	0.6 (21%)	0.76
matrixMult	0.13	8.62	7.6 (12%)	4.19 (51%)	3.22 (63%)	1.79
matrixInv	0.02	5.9	5.71 (3%)	4.84 (18%)	3.8 (36%)	0.7
insertSort	2.63	46.15	2.81 (94%)	51.54 (-10%)	2.75 (94%)	35.3
gain moyen			32 %	12 %	55%	

FIGURE 9 – Résultats des expérimentations (temps en secondes).

quelques opérations a peu d'effet. Au contraire, les cas où l'analyse flot de données arrière ne provoque (quasiment) aucun gain de performance correspondent aux cas dans lesquels beaucoup d'opérations mémoires sont effectuées : dès lors, il est de toute façon nécessaire d'observer une grande partie, si ce n'est l'intégralité, de la mémoire.

Sans surprise, en combinant les deux analyses les gains sont encore supérieurs et font gagner un ou deux ordres de magnitude au temps d'exécution, rendant la perte d'efficacité liée à l'instrumentation par *E-ACSL* supportable en pratique. On peut signaler ici que des travaux menés en parallèle pour optimiser le modèle mémoire sous-jacent utilisé par *E-ACSL* [11] font encore gagner un ordre de magnitude aux performances.

Enfin, il est difficile de nous comparer avec les autres outils de vérification dynamique, en particulier *Valgrind*. En effet, grâce à l'expressivité du langage *E-ACSL*, nous pouvons effectuer beaucoup plus de vérifications. Par exemple, nous vérifions des invariants de boucles ainsi que les propriétés fonctionnelles des opérations de tris. Ceci implique notamment le parcours de la structure résultante pour vérifier qu'elle est bien triée, éventuellement à chaque itération de boucle, ce qui est très couteux dans le cas de structures à dizaine de milliers d'éléments. Lorsque nous ne cherchons pas à vérifier des propriétés supplémentaires, nous pouvons constater que le programme instrumenté par *E-ACSL* est en général au moins aussi rapide que celui instrumenté par *Valgrind*.

6. Conclusion

Cet article a présenté deux analyses statiques présentes dans le greffon *E-ACSL* de *Frama-C*. La première est un système de types fondé sur une inférence d'intervalles ayant pour but de limiter l'utilisation des entiers *GMP* dans le code instrumenté. Grâce à ce système, de tels entiers ne sont – de fait – que rarement générés. La seconde analyse est une analyse flot de données arrière sur-approximante paramétrée par une analyse d'alias. Elle permet de limiter le nombre d'opérations mémoires à observer et, de ce fait, de limiter l'appel à la bibliothèque *C* modélisant la mémoire.

Ces deux analyses sont indispensables en pratique pour obtenir des temps d'exécution et une consommation mémoire raisonnables. On peut notamment signaler qu'AdaCore a adapté au langage *SPARK 2014*[7] l'idée du système de types présenté ici pour permettre aux utilisateurs de spécifier des propriétés sous forme mathématique sans crainte de débordement arithmétique à l'exécution, tout en minimisant le coût d'exécution.

À ce jour, la version distribuée du greffon *E-ACSL* de *Frama-C* contient des versions préliminaires de ces deux analyses statiques, tandis que l'implémentation des versions présentées dans cet article est en cours. Ces dernières permettent notamment d'améliorer encore davantage la précision des résultats présentés, et donc le temps d'exécution et la consommation mémoire des programmes générés.

7. http ://www.spark-2014.org

Remerciements Les travaux présentés dans cet article ont été partiellement financés par le programme EU-FP7 (projet STANCE, bourse 317753). Nous souhaitons remercier Yannick Moy pour des précisions sur AdaCore. Aussi, les remarques pertinentes émises par les rapporteurs anonymes ont été particulièrement utiles pour améliorer cet article.

Références

[1] S. Blazy and X. Leroy. Mechanized semantics for the Clight subset of the C language. *Journal of Automated Reasoning*, 43(3) :263–288, 2009.

[2] L. Clarke and D. Rosenblum. A historical perspective on runtime assertion checking in software development. *Software Engineering Notes*, 31(3) :25–37, 2006.

[3] L. Correnson and J. Signoles. Combining Analyses for C Program Verification. In *Formal Methods for Industrial Case Studies (FMICS'12)*, 2012.

[4] P. Cousot and R. Cousot. Abstract interpretation : a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Principles of Programming Languages (POPL'77)*, pages 238–252, 1977.

[5] P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. Frama-C : a Software Analysis Perspective. In *Software Engineering and Formal Methods (SEFM'12)*, pages 233–247. Springer, October 2012.

[6] M. Delahaye, N. Kosmatov, and J. Signoles. Common specification language for static and dynamic analysis of C programs. In *Symposium on Applied Computing (SAC'13)*, pages 1230–1235. ACM, March 2013.

[7] E. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8), 1975.

[8] C. Ellison and G. Rosu. An Executable Formal Semantics of C with Applications. In *Principles of Programming Languages (POPL'12)*, pages 533–544, 2012.

[9] A. Giorgetti, J. Groslambert, J. Julliand, and O. Kouchnarenko. Verification of class liveness properties with Java Modeling Language. *IET Software*, 2(6), 2008.

[10] P. Herms. *Certification of a Tool Chain for Deductive Program Verification.* PhD thesis, University of Paris 11, December 2012.

[11] A. Jakobsson, N. Kosmatov, and J. Signoles. Fast as a Shadow, Expressive as a Tree : Hybrid Memory Monitoring for C. In *Symposium on Applied Computing (SAC'15).* À paraître.

[12] F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. Frama-C : a Software Analysis Perspective. *Formal Aspects of Computing*, À paraître, version journal étendue de [5].

[13] N. Kosmatov, G. Petiot, and J. Signoles. An optimized memory monitoring for runtime assertion checking of C programs. In *Runtime Verification (RV'13)*, volume 8174 of *LNCS*, pages 167–182. Springer, September 2013.

[14] N. Nethercote and J. Seward. How to shadow every byte of memory used by a program. In *Virtual Execution Environments (VEE 2007)*, pages 65–74. ACM, 2007.

[15] F. Nielson, H. Nielson, and C. Hankin. *Principles of Program Analysis.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[16] B. Pierce. *Types and Programming Languages.* MIT Press, Cambridge, MA, USA, 2002.

[17] N. Revol. Introduction à l'arithmétique par intervalles. Rapport de recherche INRIA 4297, 2001.

[18] J. Signoles. *E-ACSL : Executable ANSI/ISO C Specification Language, version 1.5-4*, March 2014. `frama-c.com/download/e-acsl/e-acsl.pdf`.

[19] B. Steensgaard. Points-to Analysis in Almost Linear Time. In *the 23rd Symposium on Principles of Programming Languages (POPL'96)*, pages 32–41, New York, NY, USA, 1996. ACM.