Formalisation d'un vérificateur efficace d'assertions arithmétiques à l'exécution

Thibaut Benjamin, Félix Ridoux, Julien Signoles

Formalisation d'un vérificateur efficace d'assertions arithmétiques à l'exécution

Thibaut Benjamin[1] Félix Ridoux[1,2] Julien Signoles[1] *

[1] Université Paris-Saclay, CEA, List, Palaiseau, France
prenom.nom@cea.fr
[2] École Normale Supérieure de Rennes, Rennes, France
felix.ridoux@ens-rennes.fr

Résumé

La vérification d'assertions à l'exécution est une technique consistant à vérifier la validité d'annotations formelles pendant l'exécution d'un programme. Bien qu'ancienne, cette technique reste encore peu étudiée d'un point de vue théorique. Cet article contribue à pallier ce manque en formalisant un vérificateur d'assertions à l'exécution pour des propriétés arithmétiques entières. La principale difficulté réside dans la modélisation d'un générateur de code pour les propriétés visées qui génère du code à la fois correct et efficace. Ainsi, le code généré repose sur des entiers machines lorsque le générateur peut prouver qu'il est correct de le faire et sur une bibliothèque spécialisée dans l'arithmétique exacte, correcte mais moins efficace, dans les autres cas. Il utilise pour cela un système de types dédié. En outre, la logique considérée pour les propriétés inclue des constructions d'ordre supérieur. L'article présente également une implémentation de ce générateur de code au sein d'E-ACSL, le greffon de Frama-C dédié à la vérification d'assertions à l'exécution, ainsi qu'une première évaluation expérimentale démontrant empiriquement l'efficacité du code généré.

1 Introduction

Contexte. La vérification d'assertions à l'exécution (abbrégée en RAC [1] par la suite) est une technique de vérification formelle consistant à vérifier la validité d'annotations formelles (typiquement, des assertions) pendant l'exécution d'un programme [7]. RAC peut être vue comme une *technique de compilation* générant du code exécutable ou du code-octet à partir d'annotations formelles, soit directement pendant la compilation, soit indirectement en générant du code source qui est ensuite traduit en code exécutable à l'aide d'un compilateur classique.

Comparée aux méthodes formelles permettant de vérifier statiquement des propriétés formelles, comme la vérification de modèles [5], la vérification déductive [11] ou l'interprétation abstraite [21], RAC apporte des garanties moins fortes, car elle ne raisonne pas sur l'ensemble des traces d'exécutions du programme mais uniquement sur celle en cours d'exécution (ou, éventuellement, celles déjà exécutées). En contrepartie, elle est plus légère à mettre en œuvre pour l'utilisateur, en ne requérant en particulier aucune expertise particulière.

Bien qu'aussi ancienne que ses alternatives statiques, RAC a été nettement moins étudiée que celles-ci d'un point de vue théorique [23]. Par exemple, les auteurs de Spec#, un langage de spécification formelle pour C# permettant à la fois l'usage de RAC et de vérification déductive, indiquent que leur « vérificateur à l'exécution est simple » [2], sans fournir plus de détails sur son fonctionnement, tout en précisant néanmoins qu'il n'est pas suffisament efficace en pratique [3] [2].

*Le premier et le troisième auteurs ont été financés par le programme de Recherche et Innovation Horizon 2020 de l'Union Européenne sous l'accord numéro N° 883242, projet ENSURESEC.

1. pour *Runtime Assertion Checking*.
2. "The run-time checker is straightforward".
3. "the run-time overhead is prohibitive".

Voilà en effet tout l'enjeu de RAC : générer du code permettant de vérifier des propriétés à l'exécution, à la fois *correctement*, *i.e.*, en ne se trompant pas sur les verdicts de validité des propriétés vérifiées, et *efficacement*, *i.e.*, en minimisant les surcoûts à l'exécution, aussi bien en temps qu'en mémoire, induits par les vérifications effectuées.

Contributions. Cet article étudie ce problème en se focalisant sur les propriétés arithmétiques entières. Celles-ci sont en effet intéressantes à plus d'un titre : d'un point de vue utilisateur, elles sont fondamentales car inévitables en pratique, tandis que, d'un point de vue théorique, elles soulèvent des problèmes intéressants. En effet, les langages de spécification formelle modernes permettent exprimer ces propriétés dans $\mathbb{Z}$, l'ensemble des entiers relatifs, alors que les langages exécutables cibles reposent sur des représentations machines bornées (typiquement, des intervalles finis d'entiers au lieu de $\mathbb{Z}$). Pour RAC, il en résulte une tension entre correction et efficacité : la première requiert l'utilisation d'une bibliothèque spécialisée permettant de modéliser fidèlement $\mathbb{Z}$ et son arithmétique (par exemple, GMP [4] en C), au prix d'un surcoût non négligeable à l'exécution, tandis que la seconde requiert l'utilisation directe des représentations machines bornées, au risque d'être incorrecte.

Pour surmonter cette difficulté, nous utilisons ici un système de types dédié permettant de reposer, d'une manière correcte, sur une représentation machine bornée, lorsqu'il infère un type suffisamment précis. Ce système de types est antérieur à nos travaux [12, 13] mais a été étendu ici aux conditions ternaires et à trois nouveaux termes arithmétiques d'ordre supérieur représentant une somme $\Sigma_{i=1}^{n} f(i)$, un produit $\Pi_{i=1}^{n} f(i)$, et un dénombrement d'éléments d'un ensemble d'entiers satisfaisant une certaine propriété P. L'apport principal de nos travaux réside néanmoins en la formalisation de la phase de génération de code en présence d'un tel système, ce qui n'avait encore jamais été réalisé. Nous présentons également l'implantation qui en a été faite dans E-ACSL [24], le vérificateur d'annotations à l'exécution de Frama-C [3], une plateforme ouverte pour l'analyse de programmes C, ainsi qu'une première évaluation de l'efficacité du code généré pour les constructions d'ordre supérieur. En résumé, nos contributions sont les suivantes :

- une extension à l'ordre supérieur du système de types d'E-ACSL [12, 13] ;
- une formalisation du générateur de code reposant sur ce système de types ;
- une présentation de l'implémentation de ce générateur de code dans E-ACSL ;
- une évaluation de l'efficacité du code généré pour les constructions d'ordre supérieur.

Travaux connexes. Y. Cheon [6] s'est, le premier, intéressé à la formalisation d'un vérificateur à l'exécution lié à JML [14], un langage de spécification formelle pour Java. Il ne démontre néanmoins aucun résultat. En outre, il ne s'est pas intéressé à l'arithmétique, étant donné que, à l'époque de ces travaux (le tout début des années 2000), l'arithmétique de JML était exactement celle de Java : la traduction était donc exactement la fonction identité. On peut néanmoins mentionner le fait que notre notion de macros, présentée section 5.1 et sur laquelle se fonde notre traduction de l'arithmétique entière, est proche de sa notion de contexte introduite pour générer le code des constructions indéfinies de JML, comme 1/0.

Par la suite, H. Lehner [15] a formalisé en Coq une large partie de la sémantique de JML, tout en définissant et prouvant correct un algorithme pour le RAC d'une construction complexe (la clause `assignable` de JML, spécifiant les zones mémoires potentiellement écrites par une fonction), mais cette construction est indépendante des constructions arithmétiques.

Plus récemment, plusieurs travaux ont eu lieu autour d'E-ACSL. Nous avons déjà mentionné ceux liés à son système de types permettant de générer du code efficace pour l'arithmétique entière [12], étendue aux rationnels [13]. Ils n'abordent cependant pas le problème de la génération de code. G. Petiot a le premier, formalisé une transformation de programme proche d'E-ACSL

4. https://gmplib.org/

incluant l'arithmétique entière exacte [18], mais sans s'intéresser au problème d'optimisation. D. Ly s'est également intéressé à une telle transformation, mais centrée sur les propriétés mémoires [16]. En particulier, tous les entiers sont bornés.

Enfin, très récemmment, C. Pascutto et J. C. Filliâtre ont proposé Ortac, un vérificateur à l'exécution de propriétés formelles pour des programmes OCaml [9]. Il repose sur un mécanisme similaire à celui d'E-ACSL pour générer du code efficace pour l'arithmétique. Néanmoins, le mécanisme de génération de code n'est pas détaillé. *A fortiori*, il n'est pas formalisé.

Par ailleurs, W. Dietz, P. Li, J. Regher et V. Adve [8] proposent de tester à l'exécution si une opération entière provoque un débordement. Leur méthode pourrait être combinée à notre système de types lorsque ce dernier ne permet pas de conclure si le calcul peut ou non être fait avec des entiers machines bornés. Il est néanmoins à noter que leur solution repose explicitement sur LLVM, alors que notre système est indépendant du compilateur et du système sous-jacent.

Plan. D'abord, la section 2 présente un exemple complet, tandis que la section 3 introduit la syntaxe et la sémantique de nos langages d'étude. Ensuite, la section 4 présente notre système de types et la section 5 formalise le générateur de code. Enfin la section 6 présente sa mise en œuvre dans E-ACSL et une évaluation empirique de l'efficacité du code généré.

2 Exemple complet

Cette section introduit un exemple afin d'illustrer concrètement le problème. Considérons une machine 64-bit exécutant un programme C contenant trois variables entières a, b et n, et supposons que l'on veuille vérifier à l'exécution que la somme des carrés des nombres entre a et b est inférieure à n, autrement dit que $\Sigma_{k=a}^{b} k^2 < n$.

Il est possible pour cela d'utiliser le langage de spécification ACSL [4] afin d'écrire à l'endroit adéquat du programme l'assertion `/*@ assert \sum(a, b, \lambda integer k; k * k)< n; */` et d'utiliser le greffon E-ACSL [24] de Frama-C [3] afin de transformer l'annotation en code C à compiler avec le reste du programme. En supposant que n soit un entier de type `int` codé sur quatre octets, la Figure 1 présente des versions, très légèrement simplifiées à des fins didactiques, des deux traductions possibles effectuées par E-ACSL, en fonction du type des variables a et b (`char` codé sur un octet à gauche et `int` codé sur quatre octets à droite).

On peut remarquer que les deux versions du code généré suivent la même structure, à savoir d'abord des déclarations et initialisations des mêmes variables temporaires (sauf pour les types et pour les variables $\_\_n$ et $\_\_eq$ présentes à droite et pas à gauche), ensuite une boucle permettant d'itérer sur tous les nombres entre a et b et calculant la somme $\_\_sum$ de leurs carrés, enfin une comparaison de cette somme avec n. Néanmoins, les types des variables générées et les opérations associées diffèrent fortement.

En effet, dans la version de gauche pour laquelle a et b sont de type `char`, et donc avec des valeurs comprises entre -128 et 127, E-ACSL utilise un système de types dédié pour inférer automatiquement que la somme des carrés entre a et b est nécessairement comprise entre $256 \times (-128) \times 127 = -4\,161\,536$ et $256 \times (-128)^2 = 4\,194\,560$ car le nombre maximal de valeurs entre a et b inclus est 256 et que le plus petit (*resp.* grand) carré calculé est -128×127 (*resp.* $(-128)^2$) : il s'agit d'une sur-approximation des résultats possibles, qui peut sembler importante mais qui est néanmoins suffisante pour conclure que tous les calculs peuvent être effectués dans le type `int`, dont les valeurs sont comprises entre -2^{31} et $2^{31} - 1$, sans aucun risque de dépassement arithmétique. En conséquence, le code généré repose sur ce type de données.

Néanmoins, lorsque les variables a et b sont de type `int` comme dans la version de droite, et en suivant le même raisonnement, E-ACSL infère automatiquement que la somme des carrés est, cette fois, nécessairement comprise entre $2^{32} \times (-2^{31}) \times (2^{31} - 1) = -2^{94} + 2^{62}$ et $2^{32} \times (-2^{31})^2 =$

```c
// code si a et b sont de type char:
int __k;
int __one;
int __cond;
int __lambda;
int __sum;
__one = 1;
__cond = 0;
__lambda = 0;
__sum = 0;
__k = (int)a;
while (1) {
  __cond = __k > (int)b;
  if (__cond) break;
  else {
    __lambda = __k * __k;
    __sum += __lambda;
    __k += __one;
  }
}
__e_acsl_assert(__sum < n);
```

```c
// code si a et b sont de type int:
long __k;
long __one;
int __cond;
__mpz_t __lambda;
__mpz_t __sum;
__mpz_t __n;
int __eq;
__one = 1;
__cond = 0;
__gmpz_init_set_si(__lambda,0L);
__gmpz_init_set_si(__sum,0L);
__k = (long)a;
while (1) {
  __cond = __k > (long)b;
  if (__cond) break;
  else {
    { __mpz_t __1;
      __gmpz_init_set_si(__1,__k *
          __k);
      __gmpz_set(__lambda, __1);
      __gmpz_clear(__1); }
    __gmpz_add(__sum,__sum,__lambda);
    __k += __one;
  }
}
__gmpz_init_set_si(__n,(long)n);
__eq = __gmpz_cmp(__sum,__n);
__e_acsl_assert(__eq < 0);
__gmpz_clear(__lambda);
__gmpz_clear(__sum);
__gmpz_clear(__n);
```

FIGURE 1 – Code généré par E-ACSL pour vérifier que la somme des carrés entre a et b est inférieure à n en supposant, à gauche (resp. droite), que a et b sont de type `char` (resp. `int`).

2^{94}, ce qui inclut des entiers bien au-delà de ceux représentables par une machine 64-bit. Par conséquent, afin de ne pas risquer un dépassement arithmétique, E-ACSL choisit de reposer sur des entiers GMP en précision arbitraire, de type `__mpz_t`, pour générer le code. Ainsi, toutes les opérations sur ces entiers sont effectués *via* des appels de fonctions définies dans la bibliothèque GMP, sans compter que chaque entier GMP doit être désalloué après utilisation *via* un appel à la fonction `__gmpz_clear`. On peut aussi noter que le type utilisé pour l'indice de boucle `__k` est `long` car E-ACSL infère qu'à l'issue de la dernière itération de la boucle, cette variable peut valoir, dans le pire des cas, $2^{31} - 1 + 1 = 2^{31}$, ce qui n'est pas représentable dans le type `int` mais l'est dans le type `long` dont la plus grande valeur est $2^{63} - 1$.

Ainsi, E-ACSL génère du code reposant sur des entiers machines efficaces lorsque son système de types lui indique qu'il est toujours correct de le faire, alors qu'il utilise des entiers GMP, un ordre de magnitude plus lent, lorsque son système de types estime qu'il existe une possibilité de dépasser le plus petit ou le plus grand des entiers représentables.

3 Langages d'étude

Cette section introduit la syntaxe (section 3.1) et la sémantique (section 3.2) des langages qui serviront de support à notre formalisation. Il s'agit de versions simplifiées des langages C et ACSL, intégrant également la bibliothèque GMP.

$\langle statement \rangle$ ::= $\langle variable \rangle$ '=' $\langle expression \rangle$ affectation
| $\langle statement \rangle$ ';' $\langle statement \rangle$ séquence
| 'if(' $\langle expression \rangle$ ')' $\langle statement \rangle$ 'else' $\langle statement \rangle$ condition
| 'while(' $\langle expression \rangle$ ')' $\langle statement \rangle$ boucle
| '/*@ assert' $\langle predicate \rangle$ '; */' assertion logique
| 'assert(' $\langle expression \rangle$ ')' assertion programmatique
| $\langle mpz\ statement \rangle$ appel à GMP

$\langle expression \rangle$ ::= $\langle integer \rangle$
| $\langle variable \rangle$ variable entière
| $\langle expression \rangle \diamond \langle expression \rangle$ $\diamond \in \{$'+', '-', '*', '/'$\}$
| $\langle expression \rangle \triangleleft \langle expression \rangle$ $\triangleleft \in \{$'<', '<=', '>', '>=', '==', '!='$\}$

(a) Syntaxe du langage mini-C.

$\langle predicate \rangle$ ::= '\true'
| '\false'
| $\langle term \rangle \triangleleft \langle term \rangle$ $\triangleleft \in \{$'<', '<=', '>', '>=', '==', '!='$\}$
| '!' $\langle predicate \rangle$
| $\langle predicate \rangle$ '||' $\langle predicate \rangle$

$\langle term \rangle$ ::= $\langle integer \rangle$
| $\langle variable \rangle$ variable mini-C
| $\langle binder \rangle$ variable logique
| $\langle term \rangle \diamond \langle term \rangle$ $\diamond \in \{$'+', '-', '*', '/'$\}$
| $\langle predicate \rangle$ '?' $\langle term \rangle$ ':' $\langle term \rangle$ terme conditionel
| '\sum(' $\langle term \rangle$ ',' $\langle term \rangle$ ', \lambda' $\langle binder \rangle$ ';' $\langle term \rangle$ ')'
| '\product(' $\langle term \rangle$ ',' $\langle term \rangle$ ', \lambda' $\langle binder \rangle$ ';' $\langle term \rangle$ ')'
| '\numof(' $\langle term \rangle$ ',' $\langle term \rangle$ ', \lambda' $\langle binder \rangle$ ';' $\langle predicate \rangle$ ')'

(b) Syntaxe du langage mini-ACSL.

$\langle mpz\ statement \rangle$ ::= 'mpz_init(' $\langle variable \rangle$ ')' allocation d'un mpz
| 'mpz_set_int(' $\langle variable \rangle$ ',' $\langle expression \rangle$ ')' affectation d'un entier mini-C dans un mpz
| 'mpz_set_mpz(' $\langle variable \rangle$ ',' $\langle variable \rangle$ ')' affectation d'un mpz dans un autre
| 'mpz_clear(' $\langle variable \rangle$ ')' dé-allocation d'un mpz
| 'mpz_add(' $\langle variable \rangle$ ',' $\langle variable \rangle$ ',' $\langle variable \rangle$ ')' addition de deux mpz
| 'mpz_sub(' $\langle variable \rangle$ ',' $\langle variable \rangle$ ',' $\langle variable \rangle$ ')' soustraction de deux mpz
| 'mpz_mul(' $\langle variable \rangle$ ',' $\langle variable \rangle$ ',' $\langle variable \rangle$ ')' multiplication de deux mpz
| 'mpz_div(' $\langle variable \rangle$ ',' $\langle variable \rangle$ ',' $\langle variable \rangle$ ')' division de deux mpz
| $\langle variable \rangle$ = 'mpz_cmp('$\langle variable \rangle$ ',' $\langle variable \rangle$ ')' comparaison de deux mpz

(c) Syntaxe du langage mini-GMP.

FIGURE 2 – Syntaxe des langages d'étude.

3.1 Syntaxe

La syntaxe de nos langages d'étude est présentée dans la Figure 2. Ces langages, au nombre de trois, sont appelés mini-C, mini-ACSL et mini-GMP et sont inter-dépendants. Le premier, mini-C, est le langage de programmation et correspond à un sous-ensemble du langage C. Les instructions (*statements*) sont constituées d'affectations et des structures de contrôle les plus classiques, étendues aux assertions logiques et programmatiques ainsi qu'aux appels à GMP. Les assertions logiques correspondent aux assertions ACSL et sont celles qui doivent être traduites en code C pour être exécutées, tandis que les assertions programmatiques sont justement les assertions exécutables vers lesquelles sont traduites les assertions logiques. Elles correspondent aux appels à `__e_acsl_assert` dans la Figure 1 (lignes 21 à gauche et 28 à droite).

Les expressions, quant à elles, sont limitées aux opérateurs arithmétiques et aux comparateurs logiques classiques. Le seul type supporté est `int`. L'extension aux autres types d'entiers bornés du C, comme `unsigned long`, ne pose aucun problème théorique, mais complique inutilement le propos. L'extension aux nombres flottants, considéré dans [13], n'est pas étudiée ici, quoique supportée en pratique (voir section 6).

Le langage mini-ACSL introduit les termes et les prédicats utilisés par les assertions logiques. Il définit une logique propositionnelle dans laquelle les termes peuvent contenir des conditions ternaires, des entiers en précision arbitraire, des variables venant du monde programmatique mini-C ou liées à des lambda-expressions. Par la suite, les premières seront nommées *variables* et les dernières *lieurs*. Les lieurs sont présents dans les trois constructions spécifiques `\sum(a,b,f)`, `\product(a,b,f)` et `\numof(a,b,p)` dénotant respectivement la somme $\sum_{k=a}^{b} f(k)$, le produit

$\prod_{k=a}^{b} f(k)$ et le nombre d'élements dans l'intervalle $[a, b]$ satisfaisant le prédicat p, autrement dit un dénombrement. Par convention, et pour distinguer aisément les variables des lieurs, les premières seront notées x et les secondes seront notées ξ, éventuellement suffixées d'un indice. On peut également noter que les connecteurs logiques sont limités à la négation et à la disjonction, ce qui est suffisant pour encoder les autres. L'extension à une logique du premier ordre, considérée dans les travaux passés [12, 13], n'est pas étudiée ici pour simplifier le propos.

Le langage mini-GMP introduit les appels à la bibliothèque GMP. Les variables en argument des fonctions sont des variables GMP modélisant des entiers en précision arbitraire. La variable stockant l'entier résultant de la fonction `mpz_cmp` est une variable mini-C. Les variables GMP sont des pointeurs qui doivent être alloués et désalloués *via*, respectivement, à des appels à `mpz_init` et `clear`. Ces variables peuvent être inialisées *via* à un appel à `mpz_set_int` ou `mpz_set_mpz` en fonction du type de l'expression en argument (type `int` ou entier GMP). Les opérations autorisées sur les entiers GMP sont les quatre opérations arithmétiques usuelles et la comparaison avec `mpz_cmp`.

3.2 Sémantique

Les sémantiques dynamiques des trois langages présentés à la section précédente sont définies à l'aide d'une sémantique opérationnelle à grand pas, décrite dans la suite de cette section.

Notations. Etant donné deux ensembles X et Y, on note $f : X \rightharpoonup Y$ pour indiquer que f est une fonction partielle de X vers Y, que l'on voit comme une fonction $f : X \to Y \uplus \bot$, où l'élément $\bot$ signifie que la fonction n'est pas définie. On note $\mathrm{cod}(f)$, le co-domaine de f. Étant donné une fonction $f : X \to Y$ et deux éléments $x \in X, y \in Y$, on note $f\{x \leftarrow y\}$ la fonction qui coïncide avec f en tout point, sauf en x où l'on a $f\{x \leftarrow y\}(x) = y$. Par ailleurs, on note $\mathbb{B} = \{V, F\}$, l'ensemble des valeurs de vérité.

Une police de caractères monospace est utilisée pour représenter les opérateurs et comparateurs syntaxiques (par exemple `+` et `<` pour l'addition et l'inégalité stricte mini-C), et une police proportionnelle pour représenter les opérateurs et comparateurs sémantiques correspondant (par exemple $+$ et $<$ pour les opérateurs correspondants aux précédents). Un opérateur (*resp.* comparateur) générique est noté $\diamond$ (*resp.* $\triangleleft$) et sa version sémantique correspondante est notée $\dot{\diamond}$ (*resp.* $\dot{\triangleleft}$). Par extension, la constante entière correspondante à une constante syntaxique entière z de nos langages sera notée $\dot{z}$.

Variables, lieurs, valeurs et environnements. L'ensemble infini dénombrable des variables programmatiques de nos langages est noté $\mathcal{V}$, tandis que celui des lieurs logiques est noté $\mathcal{L}$. En outre, on note `Int` l'ensemble des valeurs possibles d'une variable de type `int` et `Mpz` l'ensemble des valeurs possibles d'une variable de type mpz. L'ensemble des valeurs est ainsi $\mathbb{V} \triangleq \mathtt{Int} \uplus \mathtt{Mpz}$, c'est-à-dire l'union disjointe de ces deux ensembles de valeurs. De plus, étant donné une constante entière syntaxique z de mini-C ou mini-ACSL, on note z^{int} et z^{mpz} les valeurs entières correspondant à z encodée respectivement dans le type `int` et dans le type `mpz`, de manière à ce qu'un élément de $\mathbb{V}$ soit nécessairement de la forme z^{int} ou z^{mpz}. On appelle unwrap : $\mathbb{V} \to \mathbb{Z}$ la fonction qui renvoie l'entier codé par une valeur. Cette fonction vérifie donc les propriétés $\mathrm{unwrap}(n^{\mathsf{mpz}}) = n$ et $\mathrm{unwrap}(n^{\mathsf{int}}) = n$.

Enfin, un environnement programmatique, noté Δ, est une fonction partielle des variables vers les valeurs, c'est-à-dire $\Delta : \mathcal{V} \rightharpoonup \mathbb{V}$. Symmétriquement, un environnement logique, noté Λ, est une fonction partielle des lieurs vers les entiers, c'est-à-dire $\Lambda : \mathcal{L} \rightharpoonup \mathbb{Z}$. Par la suite, on appelle *environnement étendu* un couple Δ, Λ constitué d'un environnement programmatique et d'un environnement logique.

$$\frac{}{\Delta,\,\Lambda \models_p \texttt{\textbackslash true} \Rightarrow V} \qquad \frac{}{\Delta,\,\Lambda \models_p \texttt{\textbackslash false} \Rightarrow F} \qquad \frac{\Delta,\,\Lambda \models_p p \Rightarrow b}{\Delta,\,\Lambda \models_p \texttt{!}p \Rightarrow \neg b} \qquad \frac{\Delta,\,\Lambda \models_p p_1 \Rightarrow V}{\Delta,\,\Lambda \models_p p_1 \texttt{ || } p_2 \Rightarrow V}$$

$$\frac{\Delta,\,\Lambda \models_p p_1 \Rightarrow F \quad \Delta,\,\Lambda \models_p p_2 \Rightarrow b}{\Delta,\,\Lambda \models_p p_1 \texttt{ || } p_2 \Rightarrow b} \qquad \frac{\Delta,\,\Lambda \models_t t_1 \Rightarrow v_1 \quad \Delta,\,\Lambda \models_t t_2 \Rightarrow v_2}{\Delta,\,\Lambda \models_p t_1 \lhd t_2 \Rightarrow v_1 \hat{\lhd} v_2}$$

(a) Sémantique des prédicats.

$$\frac{}{\Delta,\,\Lambda \models_t z \Rightarrow \dot{z}} \qquad \frac{\Delta(x) = z^{\mathsf{int}}}{\Delta,\,\Lambda \models_t x \Rightarrow z} \qquad \frac{\Lambda(\xi) = z}{\Delta,\,\Lambda \models_t \xi \Rightarrow z} \qquad \frac{\Delta,\,\Lambda \models_t t_1 \Rightarrow v_1 \quad \Delta,\,\Lambda \models_t t_2 \Rightarrow v_2}{\Delta,\,\Lambda \models_t t1 \diamond t2 \Rightarrow v_1 \hat{\diamond} v_2}$$

$$\frac{\Delta,\,\Lambda \models_p p \Rightarrow V \quad \Delta,\,\Lambda \models_t t_1 \Rightarrow v_1}{\Delta,\,\Lambda \models_p p \,?\, t_1 : t_2 \Rightarrow v_1} \qquad \frac{\Delta,\,\Lambda \models_p p \Rightarrow F \quad \Delta,\,\Lambda \models_t t_2 \Rightarrow v_2}{\Delta,\,\Lambda \models_p p \,?\, t_1 : t_2 \Rightarrow v_2}$$

$$\frac{\Delta,\,\Lambda \models_t t_1 \Rightarrow v_1 \qquad \Delta,\,\Lambda \models_t t_2 \Rightarrow v_2 \qquad \forall k \in [v_1, v_2],\, \Delta,\,\Lambda\{\xi \leftarrow k\} \models_t t_3 \Rightarrow v_3^k}{\Delta,\,\Lambda \models_t \texttt{\textbackslash sum}(t_1,\, t_2,\, \texttt{\textbackslash lambda}\,\xi;\, t_3) \Rightarrow \sum_{k=v_1}^{v_2} v_3^k}$$

$$\frac{\Delta,\,\Lambda \models_t t_1 \Rightarrow v_1 \qquad \Delta,\,\Lambda \models_t t_2 \Rightarrow v_2 \qquad \forall k \in [v_1, v_2],\, \Delta,\,\Lambda\{\xi \leftarrow k\} \models_t t_3 \Rightarrow v_3^k}{\Delta,\,\Lambda \models_t \texttt{\textbackslash product}(t_1,\, t_2,\, \texttt{\textbackslash lambda}\,\xi;\, t_3) \Rightarrow \prod_{k=v_1}^{v_2} v_3^k}$$

$$\frac{\Delta,\,\Lambda \models_t \texttt{\textbackslash sum}(t_1,\, t_2,\, \texttt{\textbackslash lambda}\,\xi;\, p \,?\, 1 : 0) \Rightarrow v}{\Delta,\,\Lambda \models_t \texttt{\textbackslash numof}(t_1,\, t_2,\, \texttt{\textbackslash lambda}\,\xi;\, p) \Rightarrow v}$$

(b) Sémantique des termes.

FIGURE 3 – Sémantique du langage mini-ACSL.

Sémantique statique. Dans un souci de simplicité, les systèmes de types des langages mini-C, mini-GMP et mini-ACSL, c'est-à-dire leurs sémantiques statiques, ne sont pas décrits. On suppose néanmoins les programmes en entrée bien typés.

Sémantique des prédicats et des termes. La Figure 3a présente la sémantique des prédicats en utilisant un jugement $\Delta, \Lambda \models_p \texttt{pred} \Rightarrow b$ exprimant le fait que, dans l'environnement étendu (Δ, Λ), le prédicat `pred` s'évalue en une valeur de vérité $b \in \mathbb{B}$. De la même manière, la Figure 3b présente la sémantique des termes, en utilisant un jugement $\Delta, \Lambda \models_t \texttt{term} \Rightarrow v$ exprimant le fait que, dans l'environnement étendu Δ, Λ, le terme `term` est évalué en l'entier $v \in \mathbb{Z}$.

La plupart de ces règles sont classiques et ne méritent pas d'explication particulière. Les deux opérateurs d'ordre supérieur `\sum` et `\product` sont évalués vers leurs équivalents mathématiques, tandis que l'opérateur de dénombrement est encodé sémantiquement comme une somme. En outre, on peut noter qu'on ignore ici, à des fins simplificatrices, le problème des valeurs indéfinies, qui résulteraient par exemple de l'évaluation de termes comme `1/0`.

Sémantique des expressions. La Figure 4a présente la sémantique des expressions en utilisant un jugement noté $\Delta \models_e \texttt{expr} \Rightarrow v$, qui exprime le fait que l'expression `expr` dans l'environnement Δ est évaluée en la valeur $v \in \mathbb{V}$. Là encore, ces règles sont usuelles, à ceci près qu'elles modélisent les débordements arithmétiques. Pour ce faire, $\min_{\texttt{int}}$ et $\max_{\texttt{int}}$ désignent respectivement le plus petit et le plus grand entier qui que l'on peut coder avec un `int`, c'est-à-dire respectivement -2^{31} et $2^{31} - 1$ en considérant l'exemple de la section 2.

Sémantique des instructions. La Figure 4b présente les règles de sémantique pour les instructions en utilisant un jugement noté $\Delta_1 \models_s \texttt{instr} \Rightarrow \Delta_2$, exprimant le fait que l'instruction `instr` est évaluée en un environnement Δ_2 à partir d'un environnement Δ_1. Cette sémantique est standard pour un tel fragment du langage C, même si elle contient une sémantique des assertions et du langage mini-GMP permettant de manipuler le type `mpz`. Concernant les assertions, on peut noter que la sémantique est dite bloquante [10], c'est-à-dire qu'une assertion logique (respectivement programmatique) ne peut être exécutée que si son prédicat en argument est évalué à V (*resp.*, son expression en argument est évaluée à une valeur entière non nulle).

$$\frac{\texttt{min\_int} \leq \dot{z} \leq \texttt{max\_int}}{\Delta \models_e z \Rightarrow \dot{z}^{\,\mathsf{int}}} \qquad \frac{\Delta(x) = z^{\,\mathsf{int}}}{\Delta \models_e x \Rightarrow z^{\,\mathsf{int}}}$$

$$\frac{\Delta \models_e e_1 \Rightarrow {z_1}^{\,\mathsf{int}} \quad \Delta \models_e e_2 \Rightarrow {z_2}^{\,\mathsf{int}} \quad \texttt{min\_int} \leq z_1 \dot{\diamond}\, z_2 \leq \texttt{max\_int}}{\Delta \models_e e_1 \diamond e2 \Rightarrow (z_1 \dot{\diamond}\, z_2)^{\,\mathsf{int}}}$$

$$\frac{\Delta \models_e e_1 \Rightarrow {z_1}^{\,\mathsf{int}} \quad \Delta \models_e e_2 \Rightarrow {z_2}^{\,\mathsf{int}} \quad z_1 \dot{\triangleleft}\, z_2}{\Delta \models_e e_1 \triangleleft\, e_2 \Rightarrow 1^{\,\mathsf{int}}} \qquad \frac{\Delta \models_e e_1 \Rightarrow {z_1}^{\,\mathsf{int}} \quad \Delta \models_e e_2 \Rightarrow {z_2}^{\,\mathsf{int}} \quad \neg(z_1 \dot{\triangleleft}\, z_2)}{\Delta \models_e e_1 \triangleleft\, e_2 \Rightarrow 0^{\,\mathsf{int}}}$$

(a) Sémantique des expressions.

$$\frac{\Delta \models_e e \Rightarrow z^{\,\mathsf{int}}}{\Delta \models_s x = e \Rightarrow \Delta\{\texttt{x} \leftarrow z^{\,\mathsf{int}}\}} \qquad \frac{\Delta_1 \models_s s_1 \Rightarrow \Delta_2 \quad \Delta_2 \models_s s_2 \Rightarrow \Delta_3}{\Delta_1 \models_s s_1; s_2 \Rightarrow \Delta_3}$$

$$\frac{\Delta \models_e e \Rightarrow 0^{\,\mathsf{int}} \quad \Delta \models_s s_2 \Rightarrow \Delta_2}{\Delta \models_e \texttt{if}\,(e)\ s_1\ \texttt{else}\ s_2 \Rightarrow \Delta_2} \qquad \frac{\Delta \models_e e \Rightarrow z^{\,\mathsf{int}} \quad z^{\,\mathsf{int}} \neq 0 \quad \Delta \models_s s_1 \Rightarrow \Delta_1}{\Delta \models_e \texttt{if}\,(e)\ s_1\ \texttt{else}\ s_2 \Rightarrow \Delta_1}$$

$$\frac{\Delta \models_e e \Rightarrow 0^{\,\mathsf{int}}}{\Delta \models_s \texttt{while}\,(e)\ s \Rightarrow \Delta} \qquad \frac{\Delta \models_e e \Rightarrow z^{\,\mathsf{int}} \quad z \neq 0 \quad \Delta \models_s s \Rightarrow \Delta_2 \quad \Delta_2 \models_s \texttt{while}\,(e)\ s \Rightarrow \Delta_3}{\Delta \models_s \texttt{while}\,(e)\ s \Rightarrow \Delta_3}$$

$$\frac{\Delta, \emptyset \models_p p \Rightarrow V}{\Delta \models_s \texttt{/*@ assert}\ p;\ \texttt{*/} \Rightarrow \Delta} \qquad \frac{\Delta \models_e e \Rightarrow z^{\,\mathsf{int}} \quad z \neq 0}{\Delta \models_s \texttt{assert}(e); \Rightarrow \Delta}$$

$$\frac{\Delta(x) = \bot}{\Delta \models_s \texttt{mpz\_init}(x); \Rightarrow \Delta\{x \leftarrow 0^{\,\mathsf{mpz}}\}} \qquad \frac{\Delta(x) \neq \bot}{\Delta \models_s \texttt{mpz\_clear}(x); \Rightarrow \Delta\{x \leftarrow \bot\}}$$

$$\frac{\Delta(x) \neq \bot \quad \Delta \models_e e \Rightarrow z^{\,\mathsf{int}}}{\Delta \models_s \texttt{mpz\_set\_int}(x,\ e); \Rightarrow \Delta\{x \leftarrow z^{\,\mathsf{mpz}}\}} \qquad \frac{\Delta(x) \neq \bot \quad \Delta(y) = z^{\,\mathsf{mpz}}}{\Delta \models_s \texttt{mpz\_set\_mpz}(x,\ y); \Rightarrow \Delta\{x \leftarrow z^{\,\mathsf{mpz}}\}}$$

$$\frac{\Delta(x) \neq \bot \quad \Delta(y) = {v_y}^{\,\mathsf{mpz}} \quad \Delta(z) = {v_z}^{\,\mathsf{mpz}}}{\Delta \models_s \texttt{mpz\_}\dot{\diamond}(x,\ y,\ z); \Rightarrow \Delta\{x \leftarrow (v_y \dot{\diamond}\, v_z)^{\,\mathsf{mpz}}\}} \qquad \frac{\Delta(y) = {v_y}^{\,\mathsf{mpz}} \quad \Delta(z) = {v_z}^{\,\mathsf{mpz}}}{\Delta \models_s x = \texttt{mpz\_cmp}(y,\ z); \Rightarrow \Delta\{x \leftarrow (\mathrm{compare}(v_y,\ v_z))^{\,\mathsf{int}}\}}$$

(b) Sémantique des instructions.

FIGURE 4 – Sémantique des langages mini-C et mini-GMP.

Concernant les instructions mini-GMP, un appel à `mpz_init` permet d'allouer une nouvelle variable `mpz` initialisée à 0, tandis que `mpz_clear` effectue l'opération inverse, à savoir désallouer son argument. Notons que nous modélisons le fait qu'une variable x est allouée dans l'environnement Δ par la propriété $\Delta(x) \neq \bot$. La fonction `mpz_set_int` (*resp.* `mpz_set_mpz`) permet d'affecter un entier de type `int` (*resp.* `mpz`) à un entier `mpz` correctement alloué, tandis que `mpz_add`, `mpz_sub`, `mpz_mul`, `mpz_div` et `mpz_cmp` permettent de, respectivement, additionner, soustraire, multiplier, diviser et comparer. Les deux premières opérations stockent leur résultat respectif dans un troisième entier `mpz` préalablement alloué. La sémantique de la comparaison, quant à elle, repose sur une fonction compare : $\mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$ définie par

$$\mathrm{compare}(x,\ y) = \begin{cases} 1 & \text{si } x > y\,; \\ 0 & \text{si } x = y\,; \\ -1 & \text{si } x < y. \end{cases}$$

Cette sémantique est plus précise que celle indiquée par la documentation de GMP, qui se contente de spécifier le signe et non la valeur exacte. Par la suite, nous n'utilisons cette fonction que pour faire des tests à 0. Cette sur-spécification n'a donc aucune incidence.

Il est important de remarquer que cette sémantique des entiers `mpz` permet de simuler le fait que le type `mpz` soit équivalent à un pointeur [5], sans avoir besoin d'intégrer explicitement les pointeurs (ou les tableaux) dans nos langages d'étude. Ce modèle simplifié des entiers `mpz`

5. En réalité, il s'agit d'un tableau à un unique élément, ce qui est sémantiquement équivalent à un pointeur si on s'affranchit des subtilités du typage du C.

est néanmoins suffisant pour exprimer, outre les propriétés sémantiques sur les valeurs entières, les propriétés relative à la bonne formation des programmes par vis-à-vis de l'allocation, y compris l'absence de fuite mémoire si chaque entier alloué avec `mpz_init` est finalement libéré en appelant `mpz_clear`. Il s'agit alors de vérifier que, à la fin de l'exécution, le programme atteint un environnement Δ tel que, pour toute variable x de type `mpz`, on a $\Delta(x) = \bot$.

4 Système de types pour une génération de code efficace

Cette section présente le système de types permettant de décider si le code généré repose sur des entiers et une arithmétique machine bornée, efficaces mais potentiellement incorrects, ou sur des entiers et une arithmétique en précision arbitraire, inefficaces mais nécessairement corrects. Ce système repose sur une inférence d'intervalles permettant de déterminer un ensemble d'entiers consécutifs (autrement dit, un intervalle) le plus petit possible contenant nécessairement l'ensemble des valeurs d'un terme. Le jugement d'évaluation est noté $\Gamma \vDash t : I$ et spécifie que les valeurs du terme t sont incluses dans l'intervalle I dans l'environnement Γ. On note également $\mathbb{I}_\Gamma(t)$ l'intervalle I résultant de cette évaluation. Si l et u sont les valeurs respectivement minimale et maximale d'un intervalle, ce dernier est noté $[l; u]$. L'union de deux intervalles $[l_1; u_1]$ et $[l_2; u_2]$ est l'intervalle $[l_1; u_1] \sqcup [l_2; u_2] \triangleq [\min(l_1, l_2); \max(u_1, u_2)]$. Par convention, un intervalle $[l; u]$ avec $l > u$ est l'intervalle vide.

La Figure 5 introduit le système de règles d'inférence définissant $\Gamma \vDash t : I$. Les cinq premières règles, déjà présentées dans [12], introduisent les cas des constantes entières, des variables arithmétiques et logiques et des opérateurs arithmétiques. On peut en particulier noter que l'intervalle d'une variable programmatique correspond à celui défini par la plus petite et la plus grande valeur représentable dans son type (nécessairement `int` dans notre contexte) : de ce fait, notre inférence d'intervalle s'abstrait du contexte du programme et reste locale à l'assertion dans laquelle le terme est évalué. La sixième règle correspond au cas de l'opérateur de condition ternaire. L'intervalle résultant est l'union des intervalles issus des deux branches de la condition. En pratique, et même si ce n'est pas présenté ici, il est possible d'optimiser ce cas en fonction de la forme de la condition. Par exemple, si cette dernière est le prédicat $\xi \geq 0$, l'intervalle associé à ξ dans l'environnement peut être réduite à ses valeurs positives lorsqu'on évalue t_1 et à ses valeur strictement négative lorsqu'on évalue t_2. Effectuer de telles réductions sur les conditions est standard, notamment en interprétation abstraite [21].

Les neuf dernières règles gèrent le cas des termes d'ordre supérieur. Les trois premières d'entre elles correspondent au cas de la somme. Dans chaque cas, l'intervalle du terme t_3 est calculé dans l'environnement Γ étendu à la variable ξ à laquelle est associé l'intervalle $[l_1; u_2]$, correspondant aux nombres inclus entre la valeur minimale de la borne inférieure et la valeur maximale de la borne supérieure. On peut noter que cet intervalle est vide si $l_1 > u_2$, c'est-à-dire si la borne inférieure est nécessairement strictement plus grande que la borne supérieure. La borne inférieure (*resp.* supérieure) de l'intervalle résultant est la borne inférieure (*resp.* supérieure) de l'intervalle du corps t_3 du terme t, multipliée par la cardinalité minimale (*resp.* *maximale*) de l'intervalle associé à ξ, modulo les problèmes de signe qui génèrent les trois différents cas en fonction du fait que l'intervalle inféré pour t_3 soit positif, négatif, ou puisse contenir 0. La cardinalité d'un intervalle est introduite par l'opérateur δ défini comme suit :

$$\delta(a, b) = \begin{cases} a - b + 1 & \text{si } a \geq b \\ 0 & \text{sinon.} \end{cases}$$

Considérons de nouveau l'exemple de la section 2, et en particulier le terme $\Sigma_{k=a}^{b} k^2$ avec a et b deux variables programmatiques de type `int`. Leur intervalle est donc le même, à savoir $[\min_{\texttt{int}}; \max_{\texttt{int}}]$, c'est-à-dire -2^{31} et $2^{31}-1$ en considérant des `int` représentés sur quatre octets.

$$\frac{}{\Gamma \vDash z : [z;z]} \quad \frac{}{\Gamma \vDash x : [\min_{\texttt{int}}; \max_{\texttt{int}}]} \quad \frac{\xi \in \Gamma}{\Gamma \vDash \xi : \Gamma(\xi)}$$

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

$$\frac{\Gamma \vDash t_1 : [l_1;u_1] \quad \Gamma \vDash t_2 : [l_2;u_2] \quad 0 \in [l_2;u_2]}{\Gamma \vDash t_1 / \ t_2 : [\min(l_1, -u_1); \max(-l_1, u_1)]} \quad \frac{\Gamma \vDash t_1 : I_1 \quad \Gamma \vDash t_2 : I_2}{\Gamma \vDash p \,?\, t_1 : t_2 : I_1 \sqcup I_2}$$

$$\frac{\Gamma \vDash t_1 : [l_1;u_1] \quad \Gamma \vDash t_2 : [l_2;u_2] \quad \Gamma\{\xi \leftarrow [l_1;u2]\} \vDash t_3 : [l_3;u_3] \quad 0 \leq l_3}{\Gamma \vDash \texttt{\textbackslash sum}(t_1,\ t_2,\ \texttt{\textbackslash lambda}\,\xi;\ t_3) : [l_3 \times \delta(l_2,u_1); u_3 \times \delta(u_2,l_1)]}$$

$$\frac{\Gamma \vDash t_1 : [l_1;u_1] \quad \Gamma \vDash t_2 : [l_2;u_2] \quad \Gamma\{\xi \leftarrow [l_1;u2]\} \vDash t_3 : [l_3;u_3] \quad l_3 < 0 \leq u_3}{\Gamma \vDash \texttt{\textbackslash sum}(t_1,\ t_2,\ \texttt{\textbackslash lambda}\,\xi;\ t_3) : [l_3 \times \delta(u_2,l_1); u_3 \times \delta(u_2,l_1)]}$$

$$\frac{\Gamma \vDash t_1 : [l_1;u_1] \quad \Gamma \vDash t_2 : [l_2;u_2] \quad \Gamma\{\xi \leftarrow [l_1;u2]\} \vDash t_3 : [l_3;u_3] \quad u_3 < 0}{\Gamma \vDash \texttt{\textbackslash sum}(t_1,\ t_2,\ \texttt{\textbackslash lambda}\,\xi;\ t_3) : [l_3 \times \delta(u_2,l_1); u_3 \times \delta(l_2,u_1)]}$$

$$\frac{\Gamma \vDash t_1 : [l_1;u_1] \quad \Gamma \vDash t_2 : [l_2;u_2] \quad \Gamma\{\xi \leftarrow [l_1;u2]\} \vDash t_3 : [l_3;u_3] \quad 0 \leq l_3}{\Gamma \vDash \texttt{\textbackslash product}(t_1,\ t_2,\ \texttt{\textbackslash lambda}\,\xi;\ t_3) : [{l_3}^{\delta(l_2,u_1)}; {u_3}^{\delta(u_2,l_1)}]}$$

$$\frac{\Gamma \vDash t_1 : [l_1;u_1] \quad \Gamma \vDash t_2 : [l_2;u_2] \quad \Gamma\{\xi \leftarrow [l_1;u2]\} \vDash t_3 : [l_3;u_3] \quad l_3 < 0 \leq u_3 \quad |l_3| \geq u_3}{\Gamma \vDash \texttt{\textbackslash product}(t_1,\ t_2,\ \texttt{\textbackslash lambda}\,\xi;\ t_3) : [{u_3}^{\delta(u_2,l_1)}; {u_3}^{\delta(u_2,l_1)}]}$$

$$\frac{\Gamma \vDash t_1 : [l_1;u_1] \quad \Gamma \vDash t_2 : [l_2;u_2] \quad \Gamma\{\xi \leftarrow [l_1;u2]\} \vDash t_3 : [l_3;u_3] \quad l_3 < 0 \leq u_3 \quad |l_3| < u_3}{\Gamma \vDash \texttt{\textbackslash product}(t_1,\ t_2,\ \texttt{\textbackslash lambda}\,\xi;\ t_3) : [l_3 \times {u_3}^{\delta(u_2,l_1)-1}; {u_3}^{\delta(u_2,l_1)}]}$$

$$\frac{\Gamma \vDash t_1 : [l_1;u_1] \quad \Gamma \vDash t_2 : [l_2;u_2] \quad \Gamma\{\xi \leftarrow [l_1;u2]\} \vDash t_3 : [l_3;u_3] \quad u_3 < 0 \quad \delta_u = \delta(u_2,l_1) \neq 1}{\Gamma \vDash \texttt{\textbackslash product}(t_1,\ t_2,\ \texttt{\textbackslash lambda}\,\xi;\ t_3) : [-((-l_3)^{\delta_u}); (-l_3)^{\delta_u - \delta_u \% 2}]}$$

$$\frac{\Gamma \vDash t_1 : [l_1;u_1] \quad \Gamma \vDash t_2 : [l_2;u_2] \quad \Gamma\{\xi \leftarrow [l_1;u2]\} \vDash t_3 : [l_3;u_3] \quad u_3 < 0 \quad \delta_u = \delta(u_2,l_1) = 1}{\Gamma \vDash \texttt{\textbackslash product}(t_1,\ t_2,\ \texttt{\textbackslash lambda}\,\xi;\ t_3) : [-((-l_3)^{\delta_u}); (-l_3)^{\delta_u}]}$$

$$\frac{\Gamma \vDash \texttt{\textbackslash sum}(t_1,\ t_2,\ \texttt{\textbackslash lambda}\,\xi;\ p \,?\, 1 : 0) : I}{\Gamma \vDash \texttt{\textbackslash numof}(t_1,\ t_2,\ \texttt{\textbackslash lambda}\,\xi;\ p) : I}$$

FIGURE 5 – Inférence d'intervalles.

L'intervalle associé à la variable logique k est donc aussi $[-2^{31}; 2^{31}-1]$, ce qui permet d'inférer que celui de k^2 est $[-2^{62}+2^{31}; 2^{62}]$ [6]. On applique donc la deuxième des trois règles de la somme pour conclure que l'intervalle résultant est $[(-2^{62}+2^{31}) \times \delta(2^{31}-1, -2^{31}), 2^{62} \times \delta(2^{31}-1, -2^{31})]$, c'est-à-dire $[-2^{94}+2^{62}; 2^{94}]$ car, ici, le résultat des deux applications de δ est exactement 2^{32}.

Les cinq règles pour le produit sont duales de celles pour la somme, mais reposent sur des puissances en lieu et place de produits pour les bornes minimale et maximale des intervalles résultants. En outre, deux cas sont optimisés *via* deux règles dédiées. Le premier survient lorsque l'intervalle du lambda-terme peut contenir 0 et que sa valeur minimale est plus petite en valeur absolue que sa valeur maximale, par exemple l'intervalle $[-3;5]$. Dans ce cas, si le nombre d'itérations maximal est n, alors une borne inférieure correcte est $-3 \times 5^{n-1}$, ce qui est meilleur que $(-5)^n$, résultant du calcul par défaut. Le second cas optimise un produit lorsque son lambda-terme est nécessairement négatif avec un nombre d'itérations impair plus grand que 1 : on sait alors que le résultat, négatif, n'est pas une borne maximale (positive) et, donc, on peut itérer une fois de moins. La dernière règle introduit l'intervalle de l'opérateur de dénombrement en le réduisant au cas de la somme à laquelle il est équivalent.

Maintenant que l'inférence d'intervalle a été présentée, nous pouvons nous concentrer sur le système de types proprement dit, introduit par deux jugements d'évalution $\Gamma \vdash t : \tau_1 \hookleftarrow \tau_2$ et $\Gamma \vdash_p p : \tau_1 \hookleftarrow \tau_2$ pour les termes et les prédicats, respectivement, et signifiant informellement

6. On peut noter au passage la perte de précision sur la multiplication, liée à la non-prise en compte des relations entre ses arguments, et plus précisément du fait qu'il s'agit du cas particulier d'un carré.

$$\frac{\Gamma \vDash z : I}{\Gamma \vdash z : \theta(I)} \qquad \frac{}{\Gamma \vdash x : \texttt{int}} \qquad \frac{}{\Gamma \vdash \xi : \theta(\Gamma(x))} \qquad \frac{\Gamma \vdash t : \tau' \quad \tau' \preccurlyeq \tau}{\Gamma \vdash t : \tau}$$

$$\frac{\Gamma \vdash t_1 : \tau' \quad \Gamma \vdash t_2 : \tau' \quad \tau = \theta(\mathbb{I}_\Gamma(t_1 \diamond t_2))}{\Gamma \vdash t_1 \diamond t_2 : \tau \hookleftarrow \tau'} \qquad \frac{\Gamma \vdash_p p : \texttt{int} \quad \Gamma \vdash t_1 : \tau \quad \Gamma \vdash t_2 : \tau}{\Gamma \vdash p\,?\,t_1 : t_2 : \tau}$$

$$\frac{\Gamma \vdash t_1 : \tau' \quad \Gamma \vdash t_2 : \tau' \quad I = \mathbb{I}_\Gamma(t_1) \sqcup \mathbb{I}_\Gamma(t_2) \quad \tau' = \theta(I) \quad \Gamma\{\xi \leftarrow I\} \vdash t_3 : \tau \quad \texttt{name} \in \{\texttt{sum}, \texttt{product}, \texttt{numof}\}}{\Gamma \vdash \backslash\texttt{name}(t_1,\, t_2,\, \backslash\texttt{lambda}\, \xi;\, t_3) : \tau \hookleftarrow \tau'}$$

$$\frac{}{\Gamma \vdash_p \backslash\texttt{true} : \texttt{int}} \qquad \frac{}{\Gamma \vdash_p \backslash\texttt{false} : \texttt{int}} \qquad \frac{\Gamma \vdash t_1 : \tau \quad \Gamma \vdash t_2 : \tau \quad \tau = \theta(\mathbb{I}_\Gamma(t_1) \sqcup \mathbb{I}_\Gamma(t_2))}{\Gamma \vdash_p t_1 \lhd t_2 : \texttt{int} \hookleftarrow \tau} \qquad \frac{\Gamma \vdash_p p_1 : \texttt{int} \quad \Gamma \vdash_p p_2 : \texttt{int}}{\Gamma \vdash_p p_1 \,||\, p_2 : \texttt{int}}$$

FIGURE 6 – Système de types.

≪ dans l'environnement Γ, le résultat de la traduction du terme t (*resp.* du prédicat p). peut être de type τ_1 et l'opération arithmétique associée doit être effectuée avec le type τ_2 ≫. Cette distinction entre types du résultat et de l'opération est nécessaire pour typer précisément les opérateurs monotones décroissants, comme la division, pour lesquels le résultat peut être petit (et donc tenir dans un `int`), même si les arguments (et notamment le dénominateur pour la division) sont grands (ce qui nécessite de calculer la division en précision arbitraire). Elle peut néanmoins être omise lorsque les deux types sont les mêmes, auquel cas on écrit juste τ au lieu de $\tau \hookleftarrow \tau$. En outre, on muni notre algèbre de types, restreinte aux types `int` et `mpz` dans cet article, d'une relation d'ordre notée $\preccurlyeq$: en complément de son caractère réflexif, elle permet de spécifier que le type `int` est plus petit que le type `mpz`.

La Figure 6 introduit les règles des deux jugements de typage. Elles utilisent notamment un type $\theta(I)$ correspondant au plus petit type pouvant représenter l'ensemble des valeurs de l'intervalle I. Il s'agit donc du type `int` lorsque I est inclus dans l'intervalle $[\min_{\texttt{int}}; \max_{\texttt{int}}]$ et du type `mpz` sinon. Ces jugements de typage sont similaires à, quoique plus simples que, ceux déjà introduits dans les travaux passés [12, 13], à l'exception des sixième et septième règles, qui permettent respectivement de déduire le type d'une condition à partir du type de ses sous-termes, et de typer les quantificateurs étendus à partir du type de leur corps, en prenant garde à introduire correctement la variable liée dans l'environnement Γ. On peut noter que la quatrième règle est une règle standard de subsomption introduisant du sous-typage [19]. Grâce à elle, notre système de types peut déduire que tout terme de type `int` peut être utilisé dans un contexte où un terme de type `mpz` est attendu. Ainsi, l'arbre ci-dessous utilise cette règle (à l'extrême droite) pour utiliser la variable n dans un contexte `mpz` dans la dérivation associée à l'exemple de la section 2, dans le cas où a, b et n sont de type `int`, et en notant Γ_k l'environnement dans lequel k est associé à l'intervalle $[-2^{31}; 2^{31} - 1]$.

$$\dfrac{\dfrac{\vdash a : \texttt{int} \quad \vdash b : \texttt{int} \qquad \dfrac{\Gamma_k \vdash k : \texttt{int} \quad \Gamma_k \vdash k : \texttt{int} \quad \texttt{mpz} = \theta([-2^{94} + 2^{62}; 2^{94}])}{\Gamma_k \vdash k * k : \texttt{mpz} \hookleftarrow \texttt{int}}}{\vdash \backslash\texttt{sum}(a,\, b,\, \backslash\texttt{lambda}\, k;\, k * k) : \texttt{mpz} \hookleftarrow \texttt{int}} \qquad \dfrac{\vdash n : \texttt{int}}{\vdash n : \texttt{mpz}}}{\vdash_p \backslash\texttt{sum}(a,\, b,\, \backslash\texttt{lambda}\, k;\, k * k) \leq n : \texttt{int} \hookleftarrow \texttt{mpz}}$$

Par la suite, on suppose que notre typeur a été exécuté sur le programme d'entrée afin d'inférer le type de chaque terme et qu'on dispose ainsi d'une fonction $\mathcal{T}(t)$ nous le fournissant.

5 Génération de code

Cette section présente la génération de code qui repose sur le système de types pour décider s'il faut utiliser des entiers et des opérations machines ou GMP. Nous restreignons cette présentation aux cas de la somme et du produit, celui du dénombrement étant un cas particulier de somme et les autres générant du code moins complexe.

Cadre d'étude. La génération de code est effectuée dans un environnement, appelé *environnement de traduction* et prenant la forme d'une fonction $\Omega : \mathcal{L} \rightharpoonup \mathcal{V}$, qui associe à chaque lieur du programme source une variable fraîche dans le programme généré. La génération de code pour un terme (*resp.* prédicat) est ainsi une fonction $[\![\cdot, \cdot]\!]$ qui associe, à un terme t (*resp.* un prédicat p), dans un environnement Ω, un morceau de code généré. Ce morceau de code est une séquence d'instructions conclue par une affectation du résultat de l'évaluation dans une variable fraîche distinguée. Cette séquence d'instruction peut éventuellement manipuler d'autres variables fraîches, contenant notamment des résultats d'évaluation intermédiaire. Afin de simplifier notre analyse, on gère séparément l'initialisation et la libération des variables de type `mpz` et la transmission du résultat. On se focalise donc sur la liste des instructions générées, hors initialisation et libération, que l'on note $[\![\Omega, t]\!]_{\text{code}}$. Par ailleurs, on note $[\![\Omega, t]\!]_{\text{decl}}$ la liste des variables fraîches de type `mpz` utilisées dans le programme $[\![\Omega, t]\!]_{\text{code}}$, et l'on note $[\![\Omega, t]\!]_{\text{res}}$ la variable fraîche distinguée contenant le résultat de l'évaluation. La façon dont les variables fraîches sont générées et déclarées (non initialisées) dans le programme relève du détail d'implémentation. On suppose donc avoir accès à autant de variables fraîches que nécessaire, chacune correctement déclarée mais non initialisée au préalable. L'accès à une telle variable x est noté $\bar{x}$ pour rappeler cet état de fait.

On définit la fonction init_var (*resp.* init_vars) qui, étant donné une variable (*resp.* un ensemble fini de variables) de type `mpz`, renvoie le code correspondant à sa (*resp.* leur) initialisation de la façon suivante :

$$\text{init\_var}(\mathtt{z}) = \mathtt{mpz\_init}(\mathtt{z});$$
$$\text{init\_vars}(\{\mathtt{z}_1; \ldots; \mathtt{z}_n\}) = \text{init\_var}(\mathtt{z}_1) \ldots \text{init\_var}(\mathtt{z}_n)$$

On définit de manière duale clear_vars(V) qui renvoie le code désallouant un ensemble V de variables de type `mpz`. Ainsi le code généré lors d'une traduction est-il le suivant :

$$[\![\Omega, t]\!] = \text{init\_vars}([\![\Omega, t]\!]_{\text{decl}})\,;\;\; [\![\Omega, t]\!]_{\text{code}}\,;\;\; \text{clear\_vars}([\![\Omega, t]\!]_{\text{decl}})\,;$$

5.1 Définitions de macros dédiées

La prise en compte combinée des types `int` et `mpz` au moment de la génération de code introduit une explosion combinatoire du nombre de cas à considérer. Considérons par exemple le terme `\product(`t_1`, `t_2`, \lambda x;`t_3`)`. Le type renvoyé par la fonction $\mathcal{T}$ pour ce terme peut-être `int` ou `mpz`, et il en va de même pour chacun des sous-termes t_1, t_2 et t_3. On a donc un total de $2 \times 2 \times 2 \times 2 = 16$ combinaisons différentes à traiter. Une approche trop directe de la génération de code serait donc très fastidieuse, d'autant plus lorsqu'on souhaite la formaliser.

Pour éviter cela, on définit un ensemble de *macros*, introduit à la Figure 7, permettant de générer une instruction en fonction des types considérés. La macro *operation* _assignment est spécialisable pour chacune des quatre opérations arithmétiques élémentaires. Dans ces définitions, Ω est l'environnement de traduction, τ est le type de la variable résultat `v`, et t est un terme arbitraire. Ces macros contiennent des cas indéfinis (marqués par l'instruction `assert false`), qui sont prohibitifs. On s'appuie sur notre traduction de code et sur le système de type, pour s'assurer de ne jamais appeler de macro dans l'un de ces cas prohibitifs.

Lemme 1 (Correction des définitions de macros)**.** *L'ensemble des macros définit à la Figure 7 génère du code ayant une sémantique bien typée et bien définie, c'est-à-dire plus précisément :*

1. *La macro* int_assignment$(\tau, \mathtt{v}, \mathtt{z})$ *assigne à la variable* $\mathtt{v}$ *de type* τ *un entier* $\mathtt{z}$ *de type* `int`*.*

$$\frac{\min_{int} \leq \dot{\mathtt{z}} \leq \max_{int}}{\Delta \models_s \text{int\_assignment}(\tau, \mathtt{v}, \mathtt{z}) \Rightarrow \Delta\{\mathtt{v} \leftarrow \dot{\mathtt{z}}^{\tau}\}}$$

```
int_assignment(τ, v, z) :=
  match τ with :
    case int :
      v = z;
    case mpz :
      mpz_set_int(v, z);

var_assignment(Ω, τ, v, t) :=
  match τ, T(t) with:
    case int, int :
      v = [[Ω, t]]res;
    case mpz, int :
      mpz_set_int(v, [[Ω, t]]res);
    case mpz, mpz :
      mpz_set_mpz(v, [[Ω, t]]res);
    case int, mpz :
      assert false;
```

```
operation _assignment(Ω, τ, v, t) :=
  match τ, T(t) with :
    case int, int :
      v = v ◇ [[Ω, t]]res;
    case mpz, mpz :
      mpz_op(v, v, [[Ω, t]]res);
    case mpz, int :
      var_assignment(Ω, mpz, x̄, t);
      mpz_op(v, v, x̄);
    case int, mpz :
      assert false;
```

```
condition(Ω, c, τ, v, t) :=
  match τ, T(t) with :
    case int, int :
      c = v <= [[Ω, t]]res;
    case mpz, mpz :
      c = mpz_cmp(v, [[Ω, t]]res);
      c = c ≤ 0
    case mpz, int
      var_assignment(Ω, mpz, x̄, t);
      c = mpz_cmp(v, x̄);
      c = c ≤ 0
    case int, mpz
      assert false
```

FIGURE 7 – *Macros* specification.

2. *La macro* var_assignment$(\Omega, \tau, \mathtt{v}, t)$ *assigne à la variable* $\mathtt{v}$ *de type* τ *la traduction du terme* t *dans l'environnement de traduction* Ω.

$$\frac{\mathcal{T}(t) \preccurlyeq \tau}{\Delta \models_s \text{var\_assignment}(\Omega, \tau, \mathtt{v}, t) \Rightarrow \Delta\{\mathtt{v} \leftarrow (\text{unwrap}(\Delta[\![\Omega, t]\!]_{\text{res}}))^{\tau}\}}$$

3. *la macro operation* _assignment$(\Omega, \tau, \mathtt{v}, t)$ *assigne à la variable* $\mathtt{v}$ *de type* τ *le résultat de l'opération appliquée sur* $\mathtt{v}$ *et la traduction du terme* t*, en supposant que* m *et* M *désignent respectivement* $-\infty$ *et* $+\infty$ *dans le cas où* $\tau =$ ***mpz****, et* $\min_{int}$*et* $\max_{int}$*dans le cas où* $\tau =$ ***int****.*

$$\frac{\Delta(\mathtt{v}) = v^{\tau} \qquad m \leq v \diamond \text{unwrap}(\Delta([\![\Omega, t]\!]_{\text{res}})) \leq M}{\Delta \models_s \textit{operation}\ \text{\_assignment}(\Omega, \tau_v, \mathtt{v}, t) \Rightarrow \Delta\{\mathtt{v} \leftarrow v \diamond (\text{unwrap}\,(\Delta([\![\Omega, t]\!]_{\text{res}})))^{\tau}\}}$$

4. *la macro condition*$(\Omega, \mathtt{c}, \tau, \mathtt{v}, t)$ *assigne à la variable* $\mathtt{c}$ *une valeur différente de 0 lorsque la valeur de* $\mathtt{v}$ *de type* τ *est plus petite que celle de la traduction de* t*, et 0 sinon.*

$$\frac{\Delta(\mathtt{v}) = v^{\tau} \quad z \neq 0 \quad v \leq \text{unwrap}(\Delta([\![\Omega, t]\!]_{\text{res}}))}{\Delta \models_s \text{condition}(\Omega, \tau, \mathtt{v}, t) \Rightarrow \Delta\{\mathtt{c} \leftarrow z^{int}\}} \qquad \frac{\Delta(\mathtt{v}) = v^{\tau} \quad v > \text{unwrap}([\![\Omega, t]\!]_{\text{res}})}{\Delta \models_s \text{condition}(\Omega, \tau, \mathtt{v}, t) \Rightarrow \Delta\{\mathtt{c} \leftarrow 0^{int}\}}$$

Notons que tous ces lemmes reposent sur les hypothèses de bonne formation de notre programme, qui assurent que dès lors que l'une des macros est appelée avec comme paramètre une variable v, alors cette variable a été correctement initialisée précédemment, ce qui implique que $\Delta(\mathtt{v})$ soit correctement défini.

5.2 Traduction des annotations en code C

Il nous reste maintenant à définir les morceaux de code correspondant à $[\![\Omega, t]\!]_{\text{code}}$ et $[\![\Omega, t]\!]_{\text{res}}$ (on peut en effet calculer $[\![\Omega, t]\!]_{\text{decl}}$ à partir de $[\![\Omega, t]\!]_{\text{code}}$). La Figure 8 les définit pour les termes mini-ACSL correspondant à une somme (colonne de gauche) et à un produit (colonne de droite). Les autres construction sont omises mais leur traduction est nettement plus simple que les deux présentées ici, tout particulièrement une fois les macros de la section 5.1 introduites. Nous faisons ici l'hypothèse que la partie non présentée est correcte.

Intéressons nous maintenant à la traduction d'une somme $t =$ `\sum(t1,t2,\lambda x;t3)`, sachant que la traduction d'un produit suit exactement le même principe. La somme est traduite

$[\![\Omega, \texttt{\textbackslash sum (t1, t2, \textbackslash lambda x; t3)}]\!]_{\text{code}} =$
  $[\![\Omega, \texttt{t1}]\!]_{\text{code}}$;
  $[\![\Omega, \texttt{t2}]\!]_{\text{code}}$;
  $[\![\Omega, 1]\!]_{\text{code}}$;
  var_assignment$(\Omega, \tau_k, \texttt{k}, \texttt{t1})$
  int_assignment$(\tau, \texttt{sum}, 0)$
  condition$(\Omega, \bar{c}, \tau_k, \texttt{k}, \texttt{t2})$
  `while` $(\bar{c})$ {
    $[\![\Omega_k, \texttt{t3}]\!]_{\text{code}}$;
    add_assignment$(\Omega_k, \tau, \texttt{sum}, \texttt{t3})$
    add_assignment$(\Omega, \tau_k, \texttt{k}, 1)$
    condition$(\Omega, \bar{c}, \tau_k, \texttt{k}, \texttt{t2})$
  }

$[\![\Omega, \texttt{\textbackslash sum(t1, t2, \textbackslash lambda x; t3)}]\!]_{\text{res}} =$
  `sum`

$[\![\Omega, \texttt{\textbackslash product (t1, t2, \textbackslash lambda x; t3)}]\!]_{\text{code}} =$
  $[\![\Omega, \texttt{t1}]\!]_{\text{code}}$;
  $[\![\Omega, \texttt{t2}]\!]_{\text{code}}$;
  $[\![\Omega, 1]\!]_{\text{code}}$;
  var_assignment$(\Omega, \tau_k, \texttt{k}, \texttt{t1})$
  int_assignment$(\tau, \texttt{product}, 1)$
  condition$(\Omega, \bar{c}, \tau_k, \texttt{k}, \texttt{t2})$
  `while` $(\bar{c})$ {
    $[\![\Omega_k, \texttt{t3}]\!]_{\text{code}}$;
    mult_assignment$(\Omega_k, \tau, \texttt{product}, \texttt{t3})$
    add_assignment$(\Omega, \tau_k, \texttt{k}, 1)$
    condition$(\Omega, \bar{c}, \tau_k, \texttt{k}, \texttt{t2})$
  }

$[\![\Omega, \texttt{\textbackslash product(t1, t2, \textbackslash lambda x; t3)}]\!]_{\text{res}} =$
  `product`

FIGURE 8 – Formalization of code generation.

en un itérateur sur toutes les valeurs entre `t1` et `t2`, *via* une boucle, afin d'effectuer l'opération `t3`. La variable `k` est la variable de contrôle de la boucle modélisant le lieur `x`. Son type est donné par $\tau_k = \mathcal{T}(\texttt{t1}) \sqcup \mathcal{T}(\texttt{t2+1})$, car `k` doit pouvoir varier librement entre `t1` et `t2+1`, qui est sa valeur en sortie de boucle. Les opérations dépendantes de `t3` dans le corps de la boucle doivent être effectuées dans l'environnement $\Omega_k = \Omega\{x \leftarrow k\}$ pour prendre en compte cette liaison. Le lecteur attentif pourra noter que le type τ_k est différent du type τ' calculé pour le lieur par le typeur à la Figure 6. En effet, τ' n'est utilisé que pour typer `t3` et ses bornes, dont les valeurs sont indépendantes de la valeur finale de `k`, à savoir `t2+1`. Dans le cas particulier où la valeur de `t2` est $\max_{\texttt{int}}{}^{\texttt{int}}$, τ_k est donc nécessairement `mpz`, alors que τ' pourrait être `int` si les termes `t1` et `t3` demeurent des entiers suffisament petits en valeur absolue. Enfin, signalons que la traduction de la constante 1 comme un terme est un artefact technique permettant de bénéficier de la définition de $[\![\Omega, 1]\!]_{\text{res}}$ dans le corps de add_assignment. La variable de type `int` ainsi générée, constamment égale à 1, est de toute façon *inlinée* par n'importe quel compilateur.

5.3 Propriétés

Nous souhaitons maintenant analyser la sémantique de notre fonction de traduction, et montrer qu'elle satisfait les bonnes propriétés que l'on attend d'un analyseur à l'exécution.

Définition 1. On dit qu'un environnement sémantique Δ_1 est inclus dans un environnement sémantique Δ_2, et l'on note $\Delta_1 \subseteq \Delta_2$, si, pour toute variable $x \in \mathcal{V}$, $\Delta_1(x) = \Delta_2(x)$ ou $\Delta_1(x) = \bot$.

Théorème 1 (Correction du générateur de code)**.** *Soit trois contextes sémantique $\Delta, \Delta_1, \Delta_2$, un contexte logique Λ et un contexte de traduction Ω. Si les variables définies dans Δ et Λ concordent avec celles définies dans Δ_1, alors la valeur de la variable résultat issue de l'évaluation de la traduction du terme t dans l'environnement résultant de cette même traduction est égale à celle de t. Plus formellement, la règle suivante est admissible :*

$$\frac{\Delta \subseteq \Delta_1 \qquad \Lambda = \text{unwrap} \circ \Delta_1 \circ \Omega \qquad \Delta, \Lambda \models_t t \Rightarrow v}{\Delta_1 \models_e [\![\Omega, t]\!] \Rightarrow \Delta_2 \qquad \Delta_2([\![\Omega, t]\!]_{\text{res}}) = v^{\mathcal{T}(t)}}$$

Théorème 2 (Transparence du générateur de code)**.** *Dans le contexte résultant de l'exécution de la traduction d'un terme, toute variable du programme initiale garde la même valeur que*

dans le contexte précédent l'exécution de cette traduction. Plus formellement, la règle suivante est admissible.

$$\frac{\Delta_1 \models_s [\![\Omega, t]\!] \Rightarrow \Delta_2}{\forall x \in \mathcal{V}, x \notin \mathrm{cod}(\Omega) \Rightarrow \Delta_1(x) = \Delta_2(x)}$$

6 Implémentation et évaluation

Cette section introduit d'abord Frama-C et E-ACSL, avant de présenter une évaluation expérimentale des travaux présentés dans cet article menée avec ces outils.

Frama-C et E-ACSL. Le mécanisme de génération de code et le système de types présentés dans cet article sont implémentés au sein du greffon E-ACSL [24] de la plateforme Frama-C [3] pour l'analyse de code C. E-ACSL [24] est un vérificateur d'assertions à l'exécution qui prend en entrée un programme C étendu avec des annotations ACSL [4] et transforme ces dernières en un code C (typiquement, ceux de la Figure 1), afin de vérifier leur validité à l'exécution. E-ACSL supporte l'intégralité du langage C pris en charge par Frama-C, ainsi qu'un très large fragment du langage ACSL [22], plus large que le spectre couvert par cet article. On peut en particulier mentionner les propriétés mémoires, les quantifications bornées, les casts, les flottants et les nombres rationnels, les propriétés portant sur plusieurs points de programme (par exemple, pour une fonction f, la post-condition `ensures G == \old(G)+1` spécifie que la valeur de `G` en sortie de f est la valeur de `G` en entrée, incrémentée de 1) et les fonctions et prédicats définis par l'utilisateur dont, expérimentalement, ceux récursifs. À part les propriétés mémoires formellement étudiées par D. Ly [16], les autres n'ont encore jamais été formalisées dans le cadre d'E-ACSL, même si les propriétés multi-états sont discutés dans [23].

Aux détails d'implémentation et à la prise en compte des autres constructions près, la traduction effectuée par E-ACSL pour le périmètre considéré dans l'article est fidèle à notre présentation. Elle utilise la bibliothèque GMP pour définir le type `mpz` et les opérations associées. Notre sémantique de mini-GMP est conforme à la spécification en langue naturelle de GMP. E-ACSL fait en outre l'hypothèse que l'implémentation de GMP est correcte vis-à-vis de sa spécification, en se contentant de la vérifier par tests. On peut néanmoins signaler que R. Rieu-Helft s'est déjà intéressé à la preuve formelle des algorithmes implémentés dans GMP [20].

Évaluation expérimentale. Pour évaluer les performances du code généré, nous avons mesuré expérimentalement son temps d'exécution sur des exemples. Afin de quantifier l'impact de l'optimisation lié au typage, nous avons comparé ces temps à ceux obtenus en désactivant cette optimisation, et en n'utilisant que des entiers GMP. Cette évaluation a été menée à partir d'une version de développement de Frama-C, commit Git 8db71ab1, disponible en ligne [7], sur une machine Linux Ubuntu dotée d'une architecture 64-bit, d'un processeur Intel Xeon 2.80 GHz 12 cœurs et de 64Go de RAM. Les résultats présentés ici ont été obtenus, en utilisant l'exécutable Hyperfine et en fixant le nombre minimum d'exécutions à 100. Afin de simuler un nombre variable d'assertions, chacune a été dans une boucle de N itérations. Par ailleurs les assertions sont des sommes ou des produits itérant sur un intervalle de taille R. Les résultats de cette évaluation expérimentale sont présentés Figure 9

On peut remarquer que, pour les faibles valeurs de N et R, la version optimisée est peu utile car les calculs sont instantanés. Néanmoins, lorsque ces valeurs croissent, la différence devient vite significative : la version non optimisée ne passe pas à l'échelle. L'impact des optimisations du système de types est meilleure pour les sommes que pour les produits, en particulier

7. https://git.frama-c.com/pub/frama-c

R \ N	100	1000	10000	100000
100	2.2	2.2	2.1	2.6
1000	2.0	1.7	2.7	2.0
10000	2.5	2.6	1.9	2.0

$\sum_{i=1}^{R} i = \frac{R(R+1)}{2}$, avec optimisation

R \ N	100	1000	10000	100000
100	1.7	4.3	27.5	269.4
1000	4.8	21.3	207.5	2064
10000	20.4	200.5	1978	-

$\sum_{i=1}^{R} i = \frac{R(R+1)}{2}$, sans optimisation

R \ N	100	1000	10000	100000
100	3.4	4.5	18.8	183
1000	7.2	62.5	620.7	6202
10000	654.2	6590	-	-

$\prod_{i=1}^{R} i \geq R$, avec optimisation

R \ N	100	1000	10000	100000
100	3.4	4.5	32.5	319
1000	7.9	73.9	734.0	7214
10000	668	6665	-	-

$\prod_{i=1}^{R} i \geq R$, sans optimisation

FIGURE 9 – Evaluation de performances (en ms).

pour les grandes valeurs R, ce qui s'explique par le fait que les produits génèrent plus rapidement des valeurs entières très grands, requérant nécessairement des entiers GMP pour assurer leur correction, dès lors que le nombre de multiplications à effectuer est important. On peut néanmoins observer un gain peu important mais néanmoins significatif lorsque le programme contient beaucoup d'annotations, ce qui s'explique par le fait que le système de types permet d'utiliser les entiers machine pour les opérations sur les bornes du produit, même si l'opération interne au lambda-terme ne peut pas être optimisé. Pour tester cette hypothèse, nous avons effectué un test similaire (non détaillé ici), mais en utilisant des bornes non représentables dans le type `int` et en itérant $N = 10000$. L'écart entre la version optimisée et celle non optimisée devient alors non significatif, confortant notre hypothèse.

Au-delà de cette évaluation expérimentale, E-ACSL a déjà été utilisé avec succès sur des cas réels, par exemple pour évaluer des propriétés numériques de systèmes réactifs synchrones [25], ou encore garantir des propriétés de sécurité sur une bibliothèque cryptographique [1] ou du code utilisé dans l'avionique [17]. Dans tous ces cas, le système de types était toujours actif, même si son impact sur l'efficacité du code généré n'a pas été spécifiquement mesuré.

7 Conclusion

Cet article a présenté une formalisation d'un générateur de code optimisé pour vérifier à l'exécution des propriétés arithmétiques entières, y compris en présence d'opérateurs d'ordre supérieur comme une somme arbitraire sur un intervalle donné. L'optimisation est guidé par un système de types dédié. Ce générateur est implémenté dans le greffon E-ACSL de Frama-C. Les premières évaluations montrent que cette optimisation est indispensable au passage à l'échelle.

Les perspectives incluent la formalisation du générateur de code pour d'autres constructions que celles présentées dans l'article, comme les nombres rationnels ou les fonctions/prédicats récursifs/inductifs. Ces derniers gagneraient aussi à être typés plus finement qu'ils ne le sont aujourd'hui. Des optimisations additionnelles pourraient en outre être implémentées pour, par exemple, limiter le nombre de variables GMP allouées ou mémoiser des calculs GMP intermédiaires effectués plusieurs fois. Enfin, la formalisation pourrait également être assistée, par exemple à l'aide de Coq, afin de pouvoir extraire un générateur de code prouvé correct.

8 Remerciements

Les auteurs remercient les relecteurs anonymes pour leurs précieux retours.

Références

[1] Gergö Barany and Julien Signoles. Hybrid Information Flow Analysis for Real-World C Code. In *Tests and Proofs (TAP)*, July 2017.

[2] Mike Barnett, Manuel Fähndrich, K. Rustan M. Leino, Peter Müller, Wolfram Schulte, and Herman Venter. Specification and Verification : The Spec# Experience. *Communications of the ACM*, 2011.

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

[4] Patrick Baudin, Pascal Cuoq, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. *ACSL : ANSI/ISO C Specification.* https://frama-c.com/download/acsl.pdf.

[5] Bernhard Beckert, Michael Kirsten, Jonas Klamroth, and Mattias Ulbrich. Modular Verification of JML Contracts Using Bounded Model Checking. In *International Symposium On Leveraging Applications of Formal Methods, Verification and Validation (ISoLA)*, 2020.

[6] Yoonsik Cheon. *A runtime assertion checker for the Java Modeling Language.* PhD thesis, Iowa State University, 2003.

[7] Lori. A. Clarke and David S. Rosenblum. A Historical Perspective on Runtime Assertion Checking in Software Development. *SIGSOFT Software Engineering Notes*, 2006.

[8] Will Dietz, Peng Li, John Regehr, and Vikram Adve. Understanding Integer Overflow in C/C++. In *International Conference on Software Engineering (ICSE)*, 2012.

[9] Jean-Christophe Filliâtre and Clément Pascutto. Ortac : Runtime Assertion Checking for OCaml (tool paper). In *International Conference on Runtime Verification (RV)*, 2021.

[10] Alain Giorgetti, Julien Groslambert, Jacques Julliand, and Olga Kouchnarenko. Verification of class liveness properties with Java Modeling Language. *Journal of IET Software*, 2008.

[11] Reiner Hähnle and Marieke Huisman. *Deductive Software Verification : From Pen-and-Paper Proofs to Industrial Tools.* 2019.

[12] Arvid Jakobsson, Nikolai Kosmatov, and Julien Signoles. Rester statique pour devenir plus rapide, plus précis et plus mince. In *Journées Francophones des Langages Applicatifs (JFLA)*, 2015.

[13] Nikolai Kosmatov, Fonenantsoa Maurica, and Julien Signoles. Efficient Runtime Assertion Checking for Properties over Mathematical Numbers. In *International Conference on Runtime Verification (RV)*, 2020.

[14] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. *JML : A Notation for Detailed Design.* 1999.

[15] Hermann Lehner. *A Formal Definition of JML in Coq and its Application to Runtime Assertion Checking.* PhD thesis, ETH Zurich, 2011.

[16] Dara Ly, Nikolai Kosmatov, Frédéric Loulergue, and Julien Signoles. Verified Runtime Assertion Checking for Memory Properties. In *International Conference on Tests and Proofs (TAP)*, 2020.

[17] Dillon Pariente and Julien Signoles. Static Analysis and Runtime Assertion Checking : Contribution to Security Counter-Measures. In *Symposium sur la Sécurité des Technologies de l'Information et des Communications (SSTIC)*, June 2017.

[18] Guillaume Petiot, Bernard Botella, Jacques Julliand, Nikolai Kosmatov, and Julien Signoles. Instrumentation of annotated C programs for test generation. In *International Conference on Source Code Analysis and Manipulation (SCAM)*, 2014.

[19] Benjamin Pierce. *Types and Programming Languages.* MIT Press, 2002.

[20] Raphaël Rieu-Helft. A Why3 Proof of GMP Algorithms. *Journal of Formalized Reasoning*, 2019.

[21] Xavier Rival and Kwangkeun Yi. *Introduction to Static Analysis : An Abstract Interpretation Perspective.* 2020.

[22] Julien Signoles. *E-ACSL. Implementation in Frama-C Plug-in E-ACSL.* http://frama-c.com/

download/e-acsl/e-acsl-implementation.pdf.

[23] Julien Signoles. The E-ACSL Perspective on Runtime Assertion Checking. In *International Workshop on Verification and mOnitoring at Runtime EXecution (VORTEX)*, 2021.

[24] Julien Signoles, Nikolai Kosmatov, and Kostyantyn Vorobyov. E-ACSL, a Runtime Verification Tool for Safety and Security of C Programs. Tool Paper. In *International Workshop on Competitions, Usability, Benchmarks, Evaluation, and Standardisation for Runtime Verification Tools (RV-CuBES)*, September 2017.

[25] Franck Védrine, Maxime Jacquemin, Nikolai Kosmatov, and Julien Signoles. Runtime Abstract Interpretation for Numerical Accuracy and Robustness. In *International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, January 2021.