Formalisation d'un vérificateur efficace d'assertions arithmétiques à l'exécution 1/2

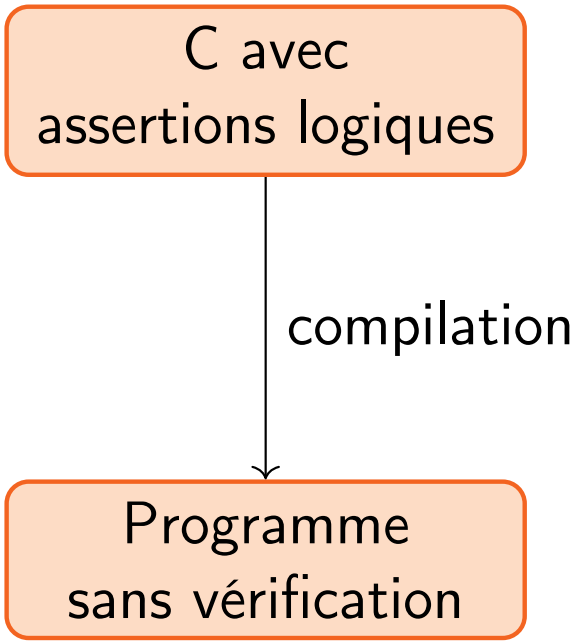
Thibaut Benjamin, Félix Ridoux, Julien Signoles

Université Paris-Saclay, CEA, List, F-91120, Palaiseau, France

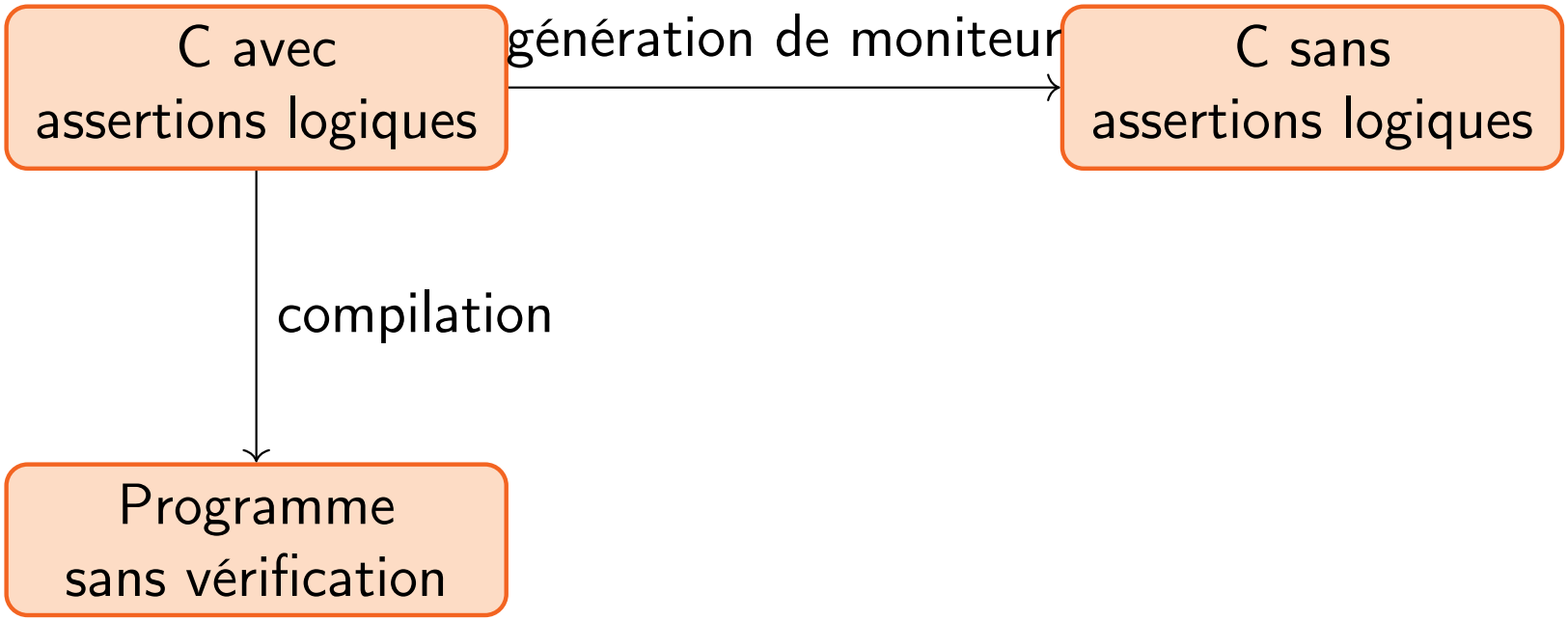
JFLA 2022

Vérification d'assertions à l'exécution

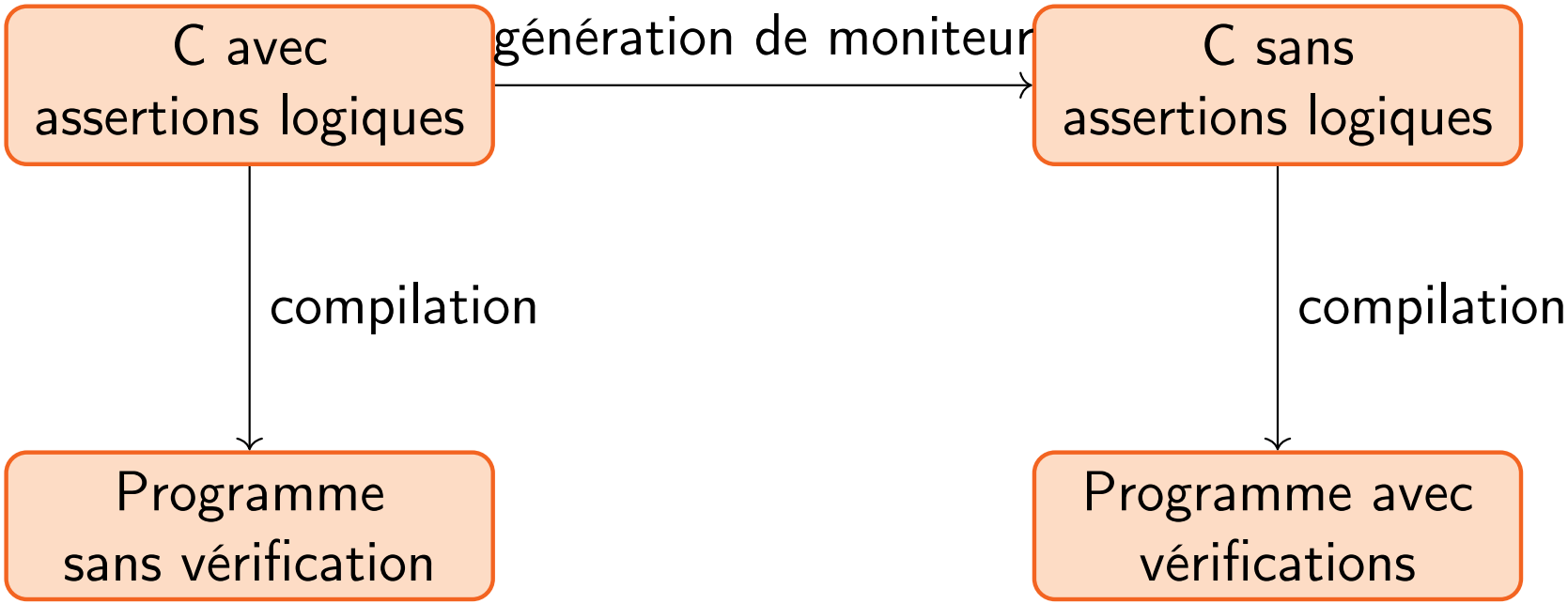
Principe de base



Principe de base



Principe de base



Exemple minimal

```
1 int main (){
2   int x = 5;
3   //@ assert x + 1 == 6;
4   return 0;
5 }
```

```
1 int main (){
2   int x = 5;
3   assert (x + 1 == 6);
4   return 0;
5 }
```

Le greffon E-ACSL

- Greffon de la platforme collaborative Frama-C
  Platforme de vérification de code Cpar collaboration de greffons utilisant plusieurs méthodes.

Le greffon E-ACSL

- Greffon de la platforme collaborative Frama-C
  Platforme de vérification de code Cpar collaboration de greffons utilisant plusieurs méthodes.
- Langage de spécifications : ACSL

Le greffon E-ACSL

- Greffon de la platforme collaborative Frama-C
  Platforme de vérification de code Cpar collaboration de greffons utilisant plusieurs méthodes.
- Langage de spécifications : ACSL
- Il supporte notamment :
  - Les assertions arithmétiques

Le greffon E-ACSL

- Greffon de la platforme collaborative Frama-C
  Platforme de vérification de code Cpar collaboration de greffons utilisant plusieurs méthodes.
- Langage de spécifications : ACSL
- Il supporte notamment :
  - Les assertions arithmétiques
  - Les quantifications sur des domaines finis

Le greffon E-ACSL

- Greffon de la platforme collaborative Frama-C
  Platforme de vérification de code Cpar collaboration de greffons utilisant plusieurs méthodes.
- Langage de spécifications : ACSL
- Il supporte notamment :
  - Les assertions arithmétiques
  - Les quantifications sur des domaines finis
  - Des vérifications de statut des pointeurs

Pour cette présentation

On s'intéresse exclusivement aux propriétés arithmétiques, comprenant :

- Les opérateurs de comparaison
  `==`, `!=`, `<`, `<=`, `>`, `>=`

Pour cette présentation

On s'intéresse exclusivement aux propriétés arithmétiques, comprenant :

- Les opérateurs de comparaison
  `==`, `!=`, `<`, `<=`, `>`, `>=`

- Les opérations arithmétiques de base
  `+`, `*`, `-`, `/`

Pour cette présentation

On s'intéresse exclusivement aux propriétés arithmétiques, comprenant :

- Les opérateurs de comparaison
  `==`, `!=`, `<`, `<=`, `>`, `>=`
- Les opérations arithmétiques de base
  `+`, `*`, `-`, `/`
- Des opérateurs quantifiés
  `\sum`, `\product`, `\numoff`

Correction ou efficacité ?

Problèmes de dépassement arithmétiques

- traduction naive

```
1 // x est un int
2 //@ assert (x+1 == 0);
```

```
1 // x est un int
2 assert (x+1 == 0);
```

Problèmes de dépassement arithmétiques

- traduction naive

```
1 // x est un int
2 //@ assert (x+1 == 0);
```

```
1 // x est un int
2 assert (x+1 == 0);
```

+ : entiers mathématiques

Problèmes de dépassement arithmétiques

- traduction naive

```
1 // x est un int
2 //@ assert (x+1 == 0);
```

$+$: entiers mathématiques

```
1 // x est un int
2 assert (x+1 == 0);
```

+ : entiers machine

Problèmes de dépassement arithmétiques

- traduction naive

```
1 // x est un int
2 //@ assert (x+1 == 0);
```

$+$: entiers mathématiques

```
1 // x est un int
2 assert (x+1 == 0);
```

+ : entiers machine

- Et si $x = 2^{31} - 1$?

Problèmes de dépassement arithmétiques

- traduction naive

```
1 // x est un int
2 //@ assert (x+1 == 0);
```

$+$: entiers mathématiques

```
1 // x est un int
2 assert (x+1 == 0);
```

+ : entiers machine

- Et si x$= 2^{31} - 1$?

  $2^{31} == 0$

Problèmes de dépassement arithmétiques

- traduction naive

```
1 // x est un int
2 //@ assert (x+1 == 0);
```

$+$: entiers mathématiques

```
1 // x est un int
2 assert (x+1 == 0);
```

+ : entiers machine

- Et si x$= 2^{31} - 1$?

$2^{31} == 0$

dépassement arithmétique

Entiers de taille arbitraire

Traduire avec la bibliothèque GMP

```
1 // x est un int
2 //@ assert (x+1 == 0);
```

```
1  // x est un int
2  mpz_t y, z, o, r;
3  mpz_init_set_si(y, x);
4  mpz_init_set_si(o, 1);
5  mpz_init(r);
6  mpz_add(r, y, o);
7  mpz_init_set_si(z, 0);
8  int c = mpz_cmp(r, 0);
9  assert (c == 0);
10 mpz_clear(y);
11 mpz_clear(z);
12 mpz_clear(o);
13 mpz_clear(r);
```

Une analyse statique

- Entiers machine : incorrects vs. entiers GMP : inefficaces
  Comment obtenir le meilleur des deux mondes ?

Une analyse statique

- Entiers machine : incorrects vs. entiers GMP : inefficaces
  Comment obtenir le meilleur des deux mondes ?

- Analyse statique d'intervalles

Une analyse statique

- Entiers machine : incorrects vs. entiers GMP : inefficaces
  Comment obtenir le meilleur des deux mondes ?

- Analyse statique d'intervalles
  - Analyse $\rightarrow$ pas de dépassement arithémtique
    Entiers machine

Une analyse statique

- Entiers machine : incorrects vs. entiers GMP : inefficaces
  Comment obtenir le meilleur des deux mondes ?

- Analyse statique d'intervalles
  - Analyse $\rightarrow$ pas de dépassement arithémtique
    Entiers machine
  - Analyse inconclusive
    Entiers GMP

Exemple : une règle pour la somme

On note δ la distance

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

Formalisation d'un vérificateur efficace d'assertions arithmétiques à l'exécution 2/2

Thibaut Benjamin, Félix Ridoux, Julien Signoles

Université Paris-Saclay, CEA, List, F-91120, Palaiseau, France

JFLA 2022

Objectifs de la génération de code

- Transformer un terme ACSL en code C de même sémantique
  - Le résultat de la transformation est dépendant du résultat du système de type.
  - La transformation doit être correct...
  - ... et transparente
- Le résultat de la génération de code pour un terme terme t dans un environnement Ω se divise en trois parties:
  - $[\![\Omega, t]\!]_{\text{decl}}$,
  - $[\![\Omega, t]\!]_{\text{code}}$,
  - $[\![\Omega, t]\!]_{\text{res}}$.

Exemple de la somme

$[\![\Omega, \backslash\texttt{sum (t1, t2, } \backslash\texttt{lambda x; t3)}]\!]_{\text{code}} =$
  $[\![\Omega, \texttt{t1}]\!]_{\text{code}}$;
  $[\![\Omega, \texttt{t2}]\!]_{\text{code}}$;
  $[\![\Omega, 1]\!]_{\text{code}}$;
  `k := t1;`
  `sum := 0;`
  `while (k <= t2) {`
    $[\![\Omega_k, \texttt{t3}]\!]_{\text{code}}$;
    `sum := sum +` $[\![\Omega, \texttt{t3}]\!]_{\text{res}}$
    `k++;`
  `}`
$[\![\Omega, \backslash\texttt{sum(t1, t2, } \backslash\texttt{lambda x; t3)}]\!]_{\text{res}} =$
  `sum`

Figure: pseudo-code de la génération de code pour la somme.

- 2*2*2*2=16 cas à formaliser...
- ...on cache cela dans des *macros*

Définition des macros

```
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: Spécification des *macros*.

Sémantique des `macros`

- Ces *macros* genèrent du code ayant une sémantique bien typée et bien définie, rendant le raisonnement sur la correction de la génération de code plus facile.
- Par exemple la sémantique de la *macro* int_assignment peut être décrite par la règle de sémantique:

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

Génération de code pour la somme

$$
\begin{array}{l}
[\![\Omega, \texttt{\textbackslash sum (t1, t2, \textbackslash lambda x; t3)}]\!]_{\text{code}} = \\
\quad [\![\Omega, \texttt{t1}]\!]_{\text{code}}\,; \\
\quad [\![\Omega, \texttt{t2}]\!]_{\text{code}}\,; \\
\quad [\![\Omega, 1]\!]_{\text{code}}\,; \\
\quad \text{var\_assignment}(\Omega, \tau_k, \texttt{k}, \texttt{t1}) \\
\quad \text{int\_assignment}(\tau, \texttt{sum}, 0) \\
\quad \text{condition}(\Omega, \bar{c}, \tau_k, \texttt{k}, \texttt{t2}) \\
\quad \texttt{while } (\bar{c}) \ \{ \\
\qquad [\![\Omega_k, \texttt{t3}]\!]_{\text{code}}\,; \\
\qquad \text{add\_assignment}(\Omega_k, \tau, \texttt{sum}, \texttt{t3}) \\
\qquad \text{add\_assignment}(\Omega, \tau_k, \texttt{k}, 1) \\
\qquad \text{condition}(\Omega, \bar{c}, \tau_k, \texttt{k}, \texttt{t2}) \\
\quad \} \\
\\
[\![\Omega, \texttt{\textbackslash sum(t1, t2, \textbackslash lambda x; t3)}]\!]_{\text{res}} = \\
\quad \texttt{sum}
\end{array}
$$

Figure: Formalisation de la génération de code pour la somme.

Implémentation et évaluation

- Implémenté au sein du greffon E-ACSL de Frama-C.
- Les évaluations expérimentales montrent que cette optimisation est indispensable au passage à l'échelle de la somme.
- Plus généralement, le principe de ce système de type permet au greffon E-ACSL d'être performant et d'être utilisé dans l'industrie.

Perspectives

- Formalisation de la génération de code pour d'autres constructions.
- Implémenter d'autres optimisations notamment pour réduire l'utilisation de GMP.
- La formalisation pourrait être assistée, par exemple à l'aide de Coq, afin de pouvoir extraire un générateur de code prouvé correct.