Formalizing an Efficient Runtime Assertion Checker for an Arithmetic Language with Functions and Predicates

Thibaut Benjamin, Julien Signoles

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

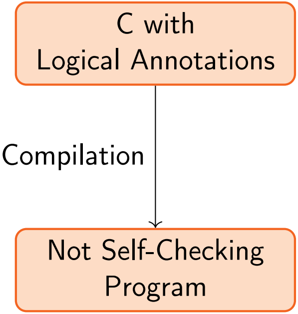
SAC-SVT 2023

RAC of Arithmetic in E-ACSL

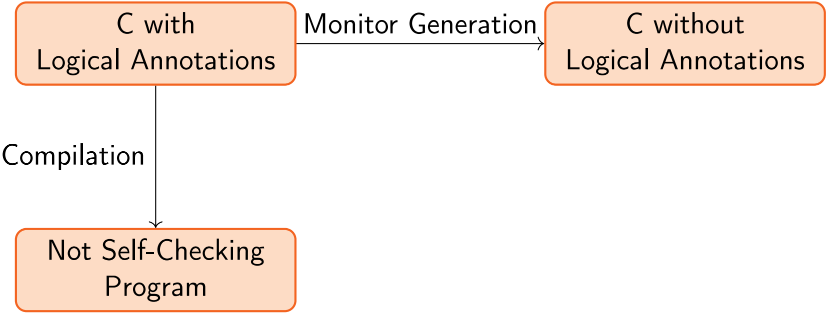
Base Principle for Inline Monitoring

C with
Logical Annotations

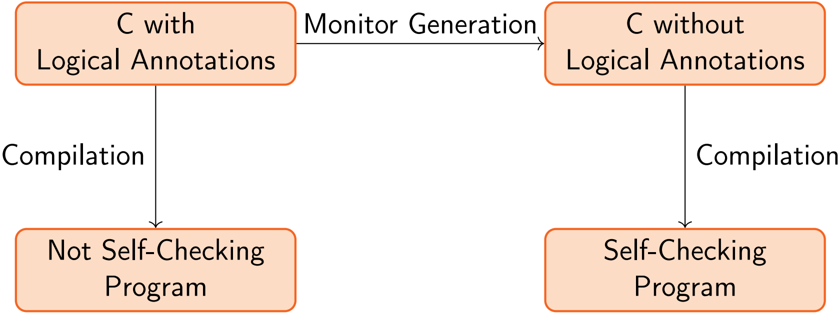
Base Principle for Inline Monitoring



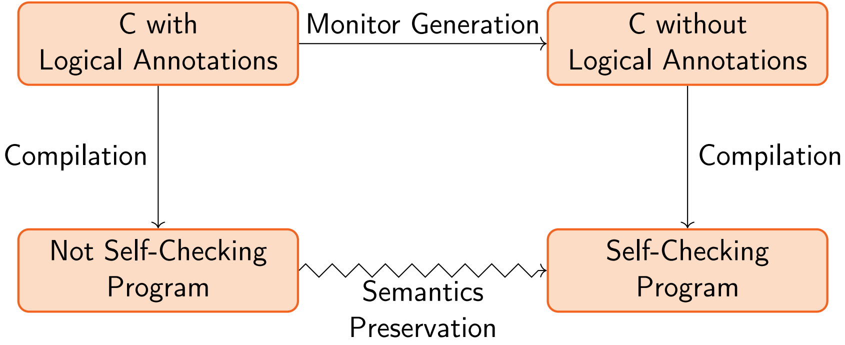
Base Principle for Inline Monitoring



Base Principle for Inline Monitoring



Base Principle for Inline Monitoring



Minimal Example

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

Minimal Example

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

$\longrightarrow$

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

The E-ACSL Plugin

- Plugin part of the Frama-C software [1]
  Collaborative platform for C code verification combining different analysis methods.

The E-ACSL Plugin

- Plugin part of the Frama-C software [1]
  Collaborative platform for C code verification combining different analysis methods.
- Specification language : ACSL [2]

The E-ACSL Plugin

- Plugin part of the Frama-C software [1]
  Collaborative platform for C code verification combining different analysis methods.
- Specification language : ACSL [2]
- E-ACSL supports in particular [3] :
  - Arithmetic Assertions

The E-ACSL Plugin

- Plugin part of the Frama-C software [1]
  Collaborative platform for C code verification combining different analysis methods.
- Specification language : ACSL [2]
- E-ACSL supports in particular [3] :
  - Arithmetic Assertions
  - User-defined (Recursive) Functions and Predicates

The E-ACSL Plugin

- Plugin part of the Frama-C software [1]
  Collaborative platform for C code verification combining different analysis methods.
- Specification language : ACSL [2]
- E-ACSL supports in particular [3] :
  - Arithmetic Assertions
  - User-defined (Recursive) Functions and Predicates
  - Quantifications on Finite Domains

The E-ACSL Plugin

- Plugin part of the Frama-C software [1]
  Collaborative platform for C code verification combining different analysis methods.
- Specification language : ACSL [2]
- E-ACSL supports in particular [3] :
  - Arithmetic Assertions
  - User-defined (Recursive) Functions and Predicates
  - Quantifications on Finite Domains
  - Pointer Status and Memory Properties

The E-ACSL Plugin

- Plugin part of the Frama-C software [1]
  Collaborative platform for C code verification combining different analysis methods.
- Specification language : ACSL [2]
- E-ACSL supports in particular [3] :
  - Arithmetic Assertions
  - User-defined (Recursive) Functions and Predicates
  - Quantifications on Finite Domains
  - Pointer Status and Memory Properties

Today

In this presentation

We are interested in the translation of a language containing the following constructs

- Comparison and arithmetic operators

  `==, !=, <, <=, >, >=`

  `+, *, -, /`

In this presentation

We are interested in the translation of a language containing the following constructs

- Comparison and arithmetic operators

  ==, !=, <, <=, >, >=

  +, *, -, /

- Conditionals

  $p\ ?\ t_1 : t_2$

In this presentation

We are interested in the translation of a language containing the following constructs

- Comparison and arithmetic operators

  `==, !=, <, <=, >, >=`

  `+, *, -, /`

- Conditionals

  $p\ ?\ t_1\ :\ t_2$

- User-defined Functions and Predicates

```
logic integer f (integer x) = t        ...     f(y)
predicate p (integer x) = b            ...     p(y)
```

Correctness vs. Efficiency

Arithmetic Overflow Issues

Naive Approach :

```
// x is an int
//@ assert (x+1 == 0);
```

→

```
// x is an int
assert (x+1 == 0);
```

Arithmetic Overflow Issues

Naive Approach :

```
// x is an int
//@ assert (x+1 == 0);
```

$\longrightarrow$

```
// x is an int
assert (x+1 == 0);
```

$+$: mathematical integers

Arithmetic Overflow Issues

Naive Approach :

```
// x is an int
//@ assert (x+1 == 0);
```

$\rightarrow$

```
// x is an int
assert (x+1 == 0);
```

$+$: mathematical integers

+ : machine integers

Arithmetic Overflow Issues

Naive Approach :

```
// x is an int
//@ assert (x+1 == 0);
```

$\rightarrow$

```
// x is an int
assert (x+1 == 0);
```

$+$: mathematical integers

+ : machine integers

What if x$= 2^{31} - 1$?

Arithmetic Overflow Issues

Naive Approach :

```
// x is an int
//@ assert (x+1 == 0);
```

$\rightarrow$

```
// x is an int
assert (x+1 == 0);
```

$+$: mathematical integers

$+$: machine integers

What if $\mathtt{x} = 2^{31} - 1$?

$2^{31} == 0$

false

Arithmetic Overflow Issues

Naive Approach :

```
// x is an int
//@ assert (x+1 == 0);
```

$\rightarrow$

```
// x is an int
assert (x+1 == 0);
```

$+$: mathematical integers

$+$: machine integers

What if x$= 2^{31} - 1$?

$2^{31} == 0$

false

Arithmetic Overflow

Undefined Behavior

Solution : Arbitrary Precision Arithmetic

A correct translation generated using the GMP library :

```
// x is an int
//@ assert (x+1 == 0);
```

```
// x is an int
mpz_t y, z, o, r;
mpz_init_set_si(y, x);
mpz_init_set_si(o, 1);
mpz_init(r);
mpz_add(r, y, o);
mpz_init_set_si(z, 0);
int c = mpz_cmp(r, 0);
assert (c == 0);
mpz_clear(y);
mpz_clear(z);
mpz_clear(o);
mpz_clear(r);
```

Static Analysis

machine integers	incorrect	efficient	simple
GMP integers	correct	inefficient	complex

Static Analysis

machine integers	incorrect	efficient	simple
GMP integers	correct	inefficient	complex

Getting the best of both worlds via a static analysis [4]

Static Analysis

machine integers	incorrect	efficient	simple
GMP integers	correct	inefficient	complex

Getting the best of both worlds via a static analysis [4]

Today : the analysis is a blackbox $\mathcal{I}$: term $\rightarrow$ interval

Static Analysis

machine integers	incorrect	efficient	simple
GMP integers	correct	inefficient	complex

Getting the best of both worlds via a static analysis [4]

Today : the analysis is a blackbox $\mathcal{I}$: term $\rightarrow$ interval

- $\mathcal{I}(t)$ is contained in the integers representable by machine
  Can safely use machine integers

Static Analysis

machine integers	incorrect	efficient	simple
GMP integers	correct	inefficient	complex

Getting the best of both worlds via a static analysis [4]

Today : the analysis is a blackbox $\mathcal{I}$: term $\rightarrow$ interval

- $\mathcal{I}(t)$ is contained in the integers representable by machine
  Can safely use machine integers
- Otherwise
  Use GMP integers in case there is an overflow

The Problem With Functions

Translating Functions

- Generate a C function that translate the ACSL function

```
logic integer p (integer x) = x + 1000000000;
```

$\downarrow$

```
int p (int x) {x + 1000000000;}
```

Translating Functions

- Generate a C function that translate the ACSL function

```
logic integer p (integer x) = x + 1000000000;
```

↓

```
int p (int x) {x + 1000000000;}
```

- Same issue with arithmetic overflows

```
/*@ assert p(1) == 1000000001; */ //OK
/*@ assert p(5000000000) > 0;   */ //Not OK
/*@ assert p(2000000000) > 0;   */ //Not OK
```

One Function Per Call-Site

- Generate a different C function for each call-site

```
/*@ assert p(1) == 1000000001; */ // -> p_1
/*@ assert p(5000000000) > 0;  */ // -> p_2
/*@ assert p(2000000000) > 0;  */ // -> p_3
```

One Function Per Call-Site

- Generate a different C function for each call-site

```
/*@ assert p(1) == 1000000001; */ // -> p_1
/*@ assert p(5000000000) > 0;  */ // -> p_2
/*@ assert p(2000000000) > 0;  */ // -> p_3
```

- Issues :

One Function Per Call-Site

- Generate a different C function for each call-site

```
/*@ assert p(1) == 1000000001; */ // -> p_1
/*@ assert p(5000000000) > 0;  */ // -> p_2
/*@ assert p(2000000000) > 0;  */ // -> p_3
```

- Issues :
  - Code duplication

```
/*@ assert p(1) == p(1); */ //Generate the
    same function twice!
```

One Function Per Call-Site

- Generate a different C function for each call-site

```
/*@ assert p(1) == 1000000001; */ // -> p_1
/*@ assert p(5000000000) > 0;  */ // -> p_2
/*@ assert p(2000000000) > 0;  */ // -> p_3
```

- Issues :
  - Code duplication

    ```
    /*@ assert p(1) == p(1); */ //Generate the same function twice!
    ```

  - Unclear for recursive functions

    ```
    /*@ logic integer f (integer x) =
          x >= 5000000000 ? 0 : f(x + 1) + 1 */
    ...
    //@ assert f(0) = 5000000000 // the recursive call escapes the type int
    ```

One Function Per Call-Context

- Generate a C function for each call-context
  A call context is the data of the interval $\mathcal{I}(t)$ for every argument of the function

One Function Per Call-Context

- Generate a C function for each call-context
  A call context is the data of the interval $\mathcal{I}(t)$ for every argument of the function
- Conservative in reuse of function

```
/*@ assert p(1) == p(1); */ //reuse the function
/*@ assert p(2) != p(1); */ //one new function
```

Dealing With Recursion

Assumption

The oracle $\mathcal{I}$ gives an interval adapted to recursive functions

Dealing With Recursion

Assumption

The oracle $\mathcal{I}$ gives an interval adapted to recursive functions

For instance :

```
/*@ logic integer f (integer x) =
      x >= 5000000000 ? 1 : f(x + 1) + 1 */
// -> interval for x+1: [1..5000000001]
...
//@ assert f(0) = 5000000000
// -> interval for 0: [0..5000000000]
```

Functional Correctness

A Complex Translation

- Starting from simple idea, the translation became non-trivial
  - Complex code using GMP
  - Subtle argument for reusing function

A Complex Translation

- Starting from simple idea, the translation became non-trivial
  - Complex code using GMP
  - Subtle argument for reusing function
- How to ensure the translation is correct ?

Formalizing the Translation

- We formalized (pen and paper style) this translation

Formalizing the Translation

- We formalized (pen and paper style) this translation
- Macro based system
  Required to avoid combinatorial blowup

Formalizing the Translation

- We formalized (pen and paper style) this translation
- Macro based system
  Required to avoid combinatorial blowup

Example :

$\mathbb{Z}\_\mathrm{assgn}(\tau_z, v, z) :=$

```
MATCH τz WITH :
  CASE int :
    v = z;
  CASE mpz :
    mpz_set_string(v,"z");
```

Assumptions

There exists a semantics the language we consider (see article) :
C programs, ACSL annotations, GMP library calls

Assumptions

There exists a semantics the language we consider (see article) :
C programs, ACSL annotations, GMP library calls

Assumption

The inference given by $\mathcal{I}$ always terminates (even on non-terminating recursive functions)

Assumptions

There exists a semantics the language we consider (see article) :
C programs, ACSL annotations, GMP library calls

Assumption

The inference given by $\mathcal{I}$ always terminates (even on non-terminating recursive functions)

Assumption

The inference given is sound : All possible semantics of t belong to $\mathcal{I}(t)$

Functional Correctness

Theorem

The translation of the subset of ACSL to the subset of C with calls to GMP library that we have defined preserves the semantics.

Functional Correctness

Theorem

The translation of the subset of ACSL to the subset of C with calls to GMP library that we have defined preserves the semantics.

Proof by induction on the different cases.

Functional Correctness

Theorem

The translation of the subset of ACSL to the subset of C with calls to GMP library that we have defined preserves the semantics.

Proof by induction on the different cases.

Avoid combinatorial blowup by proving the functional correctness of the macros independently !

What's Next ?

- Formalize and prove the oracle $\mathcal{I}$
  WIP : an article submitted !

What's Next ?

- Formalize and prove the oracle $\mathcal{I}$
  WIP : an article submitted !
- Port the formalization in Coq

What's Next ?

- Formalize and prove the oracle $\mathcal{I}$
  WIP : an article submitted !
- Port the formalization in Coq
- Study interaction between memory properties [5] and arithmetic ones

Thank you

References I

[1] P. Baudin et al. "The Dogged Pursuit of Bug-Free C Programs : The Frama-C Software Analysis Platform". In : *Communications of the ACM* (2021).

[2] P. Baudin et al. *ACSL : ANSI/ISO C Specification Language*. Rapp. tech. CEA List et Inria. url : https://frama-c.com/download/acsl.pdf.

[3] J. Signoles. *E-ACSL Version 1.18. Implementation in Frama-C Plug-in E-ACSL 26.1*. http://frama-c.com/download/e-acsl/e-acsl-implementation.pdf. 2022.

[4] N. Kosmatov, F. Maurica et J. Signoles. "Efficient Runtime Assertion Checking for Properties over Mathematical Numbers". In : *International Conference on Runtime Verification (RV)*. 2020.

References II

[5] D. Ly et al. "Verified Runtime Assertion Checking for Memory Properties". In : *International Conference on Tests and Proofs (TAP)*. 2020.