Efficient runtime assertion checking for properties over mathematical numbers

Nikolai Kosmatov, Fonenantsoa Maurica Andrianampoizinimaro, Julien Signoles

Efficient Runtime Assertion Checking for Properties over Mathematical Numbers

Nikolai Kosmatov[1,2][0000−0003−1557−2813], Fonenantsoa Maurica[1,3], and Julien Signoles[1]

[1]CEA, LIST, Software Safety and Security Laboratory, Palaiseau, France
firstname.lastname@cea.fr
[2]Thales Research & Technology, Palaiseau, France
nikolaikosmatov@gmail.com
[3]Billee, Neuilly-sur-Seine, France
firstname.lastname@billee.fr

Abstract. Runtime assertion checking is the discipline of detecting at runtime violations of program properties written as formal code annotations. These properties often include numerical properties, which may rely on either (bounded) machine representations or (unbounded) mathematical numbers. The verification of the former is easier to implement and more efficient at runtime, while the latter are more expressive and often more adequate for writing specifications. This short paper explains how the runtime assertion checker E-ACSL reconciles both approaches by presenting a type system that allows the tool to generate efficient machine-number based code when it is safe to do so, while generating arbitrary-precision code when it is necessary. This type system and the code generator not only handle integers but also rational arithmetics. As far as we know, it is the first runtime verification tool that supports the verification of properties over rational numbers.

Keywords: typing, runtime assertion checking, numerical properties, rational numbers, optimized code generation

1 Introduction

Runtime assertion checking is the discipline of detecting at runtime violations of program properties written as formal code annotations [1, 2]. This way, it allows the developer to better support testing, to make debugging easier by reducing the distance between the execution of a bug and the manifestation of its effects, to serve as executable comments about preconditions and postconditions[1], and can act as an intermediate step before applying formal proof. Its main drawback is the slowdown of the program execution. It may also lead to additional runtime failures when used incorrectly by a developer.

Formal code annotations usually express properties about program variables or/and program inputs and outputs. Most of them are mathematical properties or, at least, involve some mathematical operations. The semantics of these properties may involve either bounded machine representations (e.g., machine integers and floating-point numbers), or unbounded mathematical numbers (e.g.,

[1] https://blog.regehr.org/archives/1091

mathematical integers in $\mathbb{Z}$ and real numbers in $\mathbb{R}$). The former one allows for efficient runtime checking and remains close to the semantics of the underlying programming language, so easy to grasp for a developer. However, it is less expressive than the latter and, most often, it does not correspond to the informal specifications that writers or readers have in mind because they usually think in terms of usual mathematics [3]. It can lead to incorrect specifications [4]. Yet the latter is harder to implement and leads to less efficient code [5].

The oldest formal specification languages such as Eiffel [6] and Spec# [7] rely on a bounded semantics. JML [8] historically used this semantics but now accepts three different modes (bounded integers with modulo when overflowing, bounded integers that raises exceptions when overflowing, and unbounded integers). Spark2014 [9] also offers both bounded and unbounded integer modes. ACSL [10] and Why3 [11] rely on an unbounded semantics. Kosmatov *et al.* [12] provide a more complete comparison of Spark2014, ACSL and Why3.

This paper presents a type system that allows to rely on the unbounded mathematical semantics when writing formal specifications, while soundly using bounded machine integers most of the time when executing the code. This way, it reconciles both approaches by combining the expressiveness of the mathematical semantics and the efficiency of machine integers. It supports not only integers, but also rational numbers in $\mathbb{Q}$, thus making it possible to conveniently express and verify at runtime many useful numerical accuracy properties. This technique is implemented in E-ACSL [13], the runtime assertion checker of Frama-C [14]. This type system has also been adapted to Spark2014 by Adacore (but only for integers). As far as we know, E-ACSL is the only tool that supports runtime assertion checking of properties over rational numbers. The paper also provides initial experiments that demonstrate the gain of efficiency of E-ACSL because of its type system. To sum up, **the contributions of this paper** are threefold:

- a type system that allows a runtime assertion checker to soundly rely on machine code for evaluating mathematical integers and rational numbers;
- an implementation of this type system in E-ACSL that allows the tool to efficiently check at runtime properties over both integer and rational numbers;
- an initial evaluation of this type system to measure the gain of efficiency.

The outline of the paper is as follows. Section 2 gives a general overview and a motivating example. Section 3 introduces a small formal specification language on which the type system is designed in Section 4. Section 5 provides some insights about our evaluation and experiments before concluding in Section 6.

2 Overview and Motivating Example

To illustrate a property with mathematical numbers, consider a simple example of Fig. 1, broadly inspired by well-known numerical accuracy issues (in critical software[2] or in computations of stock prices[3]). It computes, given daily prices of a merchandise (say, in cents), its average price (as a floating-point value) over a week. The considered property states that, for a given $\varepsilon > 0$, the computed

[2] See e.g. `http://www-users.math.umn.edu/~arnold/disasters/patriot.html`
[3] See e.g. `https://en.wikipedia.org/wiki/Vancouver_Stock_Exchange`

```
1 short Mo,Tu,We,Th,Fr,Sa,Su; // daily prices in cents
2 int main{
3   ... // read recorded prices over a week
4   // compute the average price as a floating-point number
5   float avg=(Mo+Tu+We+Th+Fr+Sa+Su)/7.0;
6   // check the error is at most 0.0001
7   /*@ assert (Mo+Tu+We+Th+Fr+Sa+Su)/7.0-0.0001 ≤ avg ≤
8         (Mo+Tu+We+Th+Fr+Sa+Su)/7.0+0.0001; */
9 }
```

Fig. 1. Toy example computing an average, and an ACSL assertion on its precision.

average `avg` has a maximal (absolute) error ε with respect to the exact (ideal) average A_{id}: $A_{\text{id}} - \varepsilon \leq \texttt{avg} \leq A_{\text{id}} + \varepsilon$. For $\varepsilon = 0.0001$, this property can fail: e.g., $17293/7 \approx 2470.428571$ while the division result in a `float` is 2470.428467.

The ACSL semantics guarantees that all computations in annotations remain mathematically precise. It allows us to specify the numerical accuracy property of the result in the simplest form, as in the assertion. Indeed, unlike ACSL, in C code, a sum of integers can overflow, and a division in floating-point numbers can be rounded. In our example, the developer avoids the risk of an integer overflow by taking the prices in `short`, automatically promoted to `int` to compute the sum (line 5), but the rounding cannot be avoided.

The expression $(\texttt{Mo} + ... + \texttt{Su})/7.0$ in the assertion (lines 7–8) can have a different value than in the code (line 5) because it assumes mathematical numbers and operators in $\mathbb{Z}$ or $\mathbb{R}$. To verify the assertion at runtime, it cannot be translated into C by the same expression in machine numbers. In our example, the property translated in such a way would be always true, while the assertion can in fact fail. Precise computations in mathematical integer and rational numbers can rely e.g. on the GMP library[4] for C programs, but its usage has a cost.

The purpose of our work is twofold. First, we present a pre-analysis step that allows to identify computations for which machine numbers (of a suitable type) can be used without loss of precision. In our example, the sums on lines 7–8 can be safely computed in an `int` variable without GMP calls. Second, we add a support for rational numbers that makes it possible to easily specify and verify at runtime properties over rational numbers, including some accuracy properties.

3 Formal Specification Language

Our work is based on the E-ACSL specification language [15], derived from the ACSL specification language [10]. The differences between both languages [5] are of no importance for our work. This section introduces a small common sub-language sufficient for presenting our type system. It is rich enough to express the assertion at lines 7–8 of Fig. 1. Its formal syntax is shown in Fig. 2.

Logic statements are assertions enclosed in special comments `/*@ ...*/` that may be written before any C instruction. Assertions are typed predicates which include logical relations, comparison operators over terms, local bindings *à la* ML and bounded first-order quantifiers over integers. Terms are logic binders, C left-values (variables, pointer dereferences, array and struct accesses, etc.), mathematical constants (either integers or rationals; e.g., the constant 7.0 on

[4] https://gmplib.org/

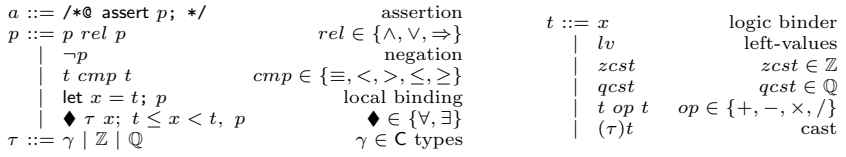

Fig. 2. Formal syntax of the specification language.

lines 7–8 in Fig. 1 is seen as a rational number because of the decimal notation)[5], numerical operators, and (explicit) casts. Terms are typed. Types are the standard C types extended with mathematical integers and rationals. The typing rules are left implicit here, but are straightforward. A numerical operation is an integer one if both arguments are integers; otherwise it is an operation over rational numbers (and the integer argument, if any, is automatically promoted to the corresponding rational number).

It is worth noting that all constants and numerical operators are over mathematical numbers (either integers in $\mathbb{Z}$, or rationals in $\mathbb{Q}$ depending on the context). C integers and floating-point values (that come from C left values) are implicitly coerced to their mathematical counterparts. For the sake of simplicity, we assume no `NaN`, `-0.0` nor $\pm\infty$ values, as well as no runtime errors when evaluating C left-values. In practice, the necessary code for checking the absence of runtime errors is generated by an independent mechanism [5, 16].

4 Type System

Preamble. The illustrative examples of this section assume a toy architecture that supports a set of C types limited to {`char`, `int`, `float`} in which the values of type `char` are included in the interval $[-32; 31]$, while those of type `int` are included in $[-128; 127]$. The possible floating-point values are left unspecified.

Illustrative examples. Consider the assertion `/*@ assert` $x + 1 \leq 127$`; */` with x of type `int`. When $x = 127$, it is invalid. Yet, using machine-integer code for evaluating it would be unsound since $x + 1$ would overflow. To circumvent this issue, a code generator should rely on a dedicated mathematical library such as GMP in C. For instance, the E-ACSL runtime assertion checker [13] of Frama-C [14] would generate the (slightly simplified) code of Fig. 3.

While sound, the generated arbitrary-precision code is not as efficient as machine-number based code, while not being always necessary. Consider for instance the assertion `/*@ assert` $c + 1 \equiv 0$`; */` with c of type `char`: it would be more efficient and fully sound to use machine-number based code for evaluating the assertion since $c + 1$ cannot overflow if computed over type `int` (the values of c vary from -32 to 31, thus the values of $c + 1$ are included in $[-128; 127]$).

Based on (over-approximated) intervals of possible values of integer-valued expressions, our type system decides whether the code generator can soundly rely on machine-number based code. On our small examples, it infers that the type of $x + 1$ should be $\mathbb{Z}$ because its possible values vary from -128 to 128,

[5] E-ACSL also supports floating-point constants such as $0.1f$ but they are excluded here for the sake of simplicity.

```
{ /* declaring temporary variables */
  mpz_t __e_acsl_x, __e_acsl, __e_acsl_add, __e_acsl_2;
  int __e_acsl_le;
  /* computing x+1 */
  __gmpz_init_set_si(__e_acsl_x, x);
  __gmpz_init_set_si(__e_acsl, 1);
  __gmpz_init(__e_acsl_add);
  __gmpz_add(__e_acsl_add, __e_acsl_x, __e_acsl);
  /* comparing x+1 and 127 */
  __gmpz_init_set_si(__e_acsl_2, 127);
  __e_acsl_le = __gmpz_cmp(__e_acsl_add, __e_acsl_2);
  e_acsl_assert(__e_acsl_le ≤ 0);
  /* freeing the allocated GMP numbers */
  __gmpz_clear(__e_acsl_x);
  __gmpz_clear(__e_acsl);
  __gmpz_clear(__e_acsl_add);
  __gmpz_clear(__e_acsl_2); }
```

Fig. 3. GMP code generated by E-ACSL for `/*@ assert` $x + 1 \leq 127$`; */` on a toy architecture. The comments have been manually inserted for readibility.

so does not fit in any existing type of our toy architecture, while the type of $c + 1$ can be `int` because its possible values vary from -32 to 32. The domains of values are computed thanks to a simple interval analysis. In most cases, using a machine-number based code instead of arbitrary-precision code is possible.

Our type system also supports rational arithmetics. Yet, our type system allows to optimize only comparisons between floating-point variables (e.g., $f \leq g$) but no rational operations. Indeed, an interval-based reasoning does not allow optimizing rational arithmetics (e.g., $f+1.$) by floating-point operations without being unsound (as any non-singular interval of rationals contains an infinite number of values non-representable as floating-point numbers). It explains why the rational extension of our type system does not rely on interval arithmetics: it directly infers either a floating-point type for variables and comparisons between them, or type $\mathbb{Q}$ for any other rational number or operator.

Language type system. We assume the existence of a type system at the level of the formal specification language: $\Sigma(t)$ denotes the type of a term t and the primitive isinteger(t) (resp., isfloat(t)) is true if and only if $\Sigma(t)$ is a subtype of $\mathbb{Z}$ (resp., a floating-point type). The relation $\preccurlyeq_\tau$ is the subtyping relation (expressing that all values of one type are also values of the other). On our illustrative examples, $\Sigma(x) =$ `int` and $\Sigma(c) =$ `char` because x (resp. c) is an `int` (resp. a `char`), while $\Sigma(x+1) = \Sigma(c+1) = \mathbb{Z}$ since any logical integer operation relies on an unbounded semantics. Furthermore, `char` $\preccurlyeq_\tau$ `int` $\preccurlyeq_\tau \mathbb{Z}$.

Integer intervals. We consider (unbounded) integer intervals with partial order $\preccurlyeq_I$. Let $\emptyset$ be the empty interval, $\mathbb{T}(I)$ be the smallest C integral type containing interval I (e.g., `int` if $I = [18; 42]$ since $I \not\subseteq [-32; 31]$), or $\mathbb{Z}$ otherwise, and $\mathbb{I}(t)$ be an interval that contains all the possible values of the term t. On our illustrative examples, $\mathbb{I}(x + 1) = [-128; 128]$ and $\mathbb{I}(c + 1) = [-32; 32]$, thus $\mathbb{T}(x+1) = \mathbb{Z}$ and $\mathbb{T}(c+1) =$ `int`. In practice, E-ACSL relies on a simple syntactic type-based inference system to compute $\mathbb{I}(t)$ [17].

Kinds. Kinds extend integer intervals to floating-point and rational numbers. They are required since integer arithmetics and rational arithmetics should re-

main separated: as already explained, integer type inference relies on interval arithmetics, while it is not the case for rationals. Kinds define a lattice structure. They are the core information used by our type system. More formally, let $(\mathbb{K}, \preccurlyeq)$ be the lattice of kinds defined as follows:

$$\begin{array}{lll} \mathbb{K} ::= \mathcal{Z}\ I & \text{an integer interval } I & \mathcal{Z}\ I_1 \preccurlyeq \mathcal{Z}\ I_2 \iff I_1 \preccurlyeq_I I_2 \\ \mid\ \mathcal{F}\ \gamma & \text{a floating-point type } \gamma & \mathcal{F}\ \gamma_1 \preccurlyeq \mathcal{F}\ \gamma_2 \iff \gamma_1 \preccurlyeq_\tau \gamma_2 \\ \mid\ \mathcal{Q} & \text{the set of rationals } \mathbb{Q} & \mathcal{Z}\ I \preccurlyeq \mathcal{F}\ \gamma \iff \mathbb{T}(I) \preccurlyeq_\tau \gamma \\ & & K \preccurlyeq \mathcal{Q} \quad (\text{for all } K \in \mathbb{K}). \end{array}$$

The kind $\mathcal{Z}\ \emptyset$ (resp., $\mathcal{Q}$) is the minimum (resp., maximum) element of the lattice. Let $\sqcup$ (resp., $\sqcap$) denote the union (resp., intersection) over kinds induced by their lattice structure. The kind of a term t, denoted $\kappa(t)$, and the type of a kind k, denoted $\theta(k)$, are defined as follows:

$$\begin{array}{ll} \kappa(t) = \mathcal{Z}\ \mathbb{I}(t) & \text{if } \text{isinteger}(t) \\ \kappa(t) = \mathcal{F}\ \Sigma(t) & \text{if } \text{isfloat}(t) \\ \kappa(t) = \mathcal{Q} & \text{if } \neg\,\text{isfloat}(t) \end{array} \qquad \begin{array}{l} \theta(\mathcal{Z}\ I) = \mathbb{T}(I) \\ \theta(\mathcal{F}\ \tau) = \tau \\ \theta(\mathcal{Q}) = \mathbb{Q}. \end{array}$$

While we will use an integer interval I for soundly representing a range of machine-integer values as soon as $\mathbb{T}(I)$ is not $\mathbb{Z}$, it would be unsound to use a non-singular rational interval R to do the same for a range of floating-point values since R contains (an infinite number of) rationals that are not representable as floating-point values. The operator κ naturally extends from terms to types. The operator θ converts a kind to a type. For integers, it means converting intervals to types, while, for rationals, it means choosing between a floating-point types and rationals. On our illustrative examples, one gets:

$$\begin{array}{rl} \theta(\kappa(x+1)) = \theta(\mathcal{Z}\ [-128;128]) & = \mathbb{Z} \\ \theta(\kappa(c+1)) = \theta(\mathcal{Z}\ [-32;32]) & = \texttt{int} \\ \theta(\kappa(f+1.)) = \theta(\mathcal{Q}) & = \mathbb{Q}. \end{array}$$

Type system. Fig. 4 presents the type system. An earlier version limited to integers has already been published in French [17]. A type judgment, written $\Gamma \vdash t : \tau_1 \hookleftarrow \tau_2$ for terms (resp., $\Gamma \vdash_p p : \tau_1 \hookleftarrow \tau_2$ for predicates), means "in the typing environment Γ, the C expression generated for t (resp., p) may soundly have type τ_1, but, in the case of an operator (resp., a comparison), it must be computed over type τ_2". The type τ_2 is omitted when irrelevant (e.g. for constants). Actually, it may only differ from τ_1 for comparisons and decreasing arithmetic operators (the division "/" in our formal specification language). Predicates return an `int`. For instance, assuming two variables x and y of type int, the term $x/(y+1)$ requires GMP code because $y+1$ does not fit into any C type of our toy architecture. However, its result fits into an int, so it may soundly be compared to 42 with the usual C equality. Therefore, its type is $\texttt{int} \hookleftarrow \mathbb{Z}$. Fig. 5 details the derivation tree of $x/(y+1) \equiv 42$ (with both x and y of type `int`) and $f - 0.1 \leq g$ (with both f and g of type `float`).

A constant is evaluated within the smallest possible type with respect to its value (rule [Cst]), but this type is actually never more precise than `int` (e.g., never `char`). This optimization avoids superfluous casts in the generated code,

$$\frac{}{\Gamma \vdash cst : \theta(\kappa(cst) \sqcup \kappa(\mathsf{int}))}\,[\textsc{Cst}] \qquad \frac{}{\Gamma \vdash lv : \Sigma(lv)}\,[\textsc{Lv}] \qquad \frac{}{\Gamma \vdash x : \Gamma(x)}\,[\textsc{Bind}]$$

$$\frac{\Gamma \vdash t : \tau_t \quad \tau' = \theta(\kappa((\tau)t))}{\Gamma \vdash (\tau)t : \tau'}\,[\textsc{Cast}] \qquad \frac{\tau = \theta(\kappa(t_1) \sqcup \kappa(t_2) \sqcup \kappa(t_1 \ \mathsf{op}\ t_2)) \quad \Gamma \vdash t_1 : \tau \quad \Gamma \vdash t_2 : \tau}{\Gamma \vdash t_1 \ \mathsf{op}\ t_2 : \tau \hookleftarrow \tau}\,[\textsc{Op}]$$

$$\frac{\Gamma \vdash t : \tau' \quad \tau' \preccurlyeq_\tau \tau}{\Gamma \vdash t : \tau}\,[\textsc{Sub}] \qquad \frac{\Gamma \vdash t : \tau' \hookleftarrow \tau' \quad \tau \prec_\tau \tau' \quad \theta(\kappa(t)) \preccurlyeq_\tau \tau}{\Gamma \vdash t : \tau \hookleftarrow \tau'}\,[\downarrow]$$

$$\frac{\Gamma \vdash_p p_1 : \mathsf{int} \quad \vdash_p p_2 : \mathsf{int}}{\Gamma \vdash_p p_1 \ rel \ p_2 : \mathsf{int}}\,[\textsc{Rel}] \qquad \frac{\Gamma \vdash_p p : \mathsf{int}}{\Gamma \vdash_p \neg p \ : \mathsf{int}}\,[\textsc{Neg}]$$

$$\frac{\tau = \theta(\kappa(t_1) \sqcup \kappa(t_2)) \quad \Gamma \vdash t_1 : \tau \quad \Gamma \vdash t_2 : \tau}{\Gamma \vdash_p t_1 \ cmp \ t_2 \ : \mathsf{int} \hookleftarrow \tau}\,[\textsc{Cmp}] \qquad \frac{\Gamma \vdash t : \tau \quad \Gamma, x : \tau \vdash_p p : \mathsf{int}}{\Gamma \vdash_p \mathsf{let}\ x = t;\ p \ : \mathsf{int}}\,[\textsc{Let}]$$

$$\frac{\begin{array}{c}\tau' = \theta(\kappa(t_1) \sqcup \kappa(t_2)) \\ \Gamma \vdash t_1 : \tau' \quad \Gamma \vdash t_2 : \tau' \quad \Gamma, x : \theta(\kappa(\tau) \sqcap \kappa(\tau')) \vdash_p p : \mathsf{int}\end{array}}{\Gamma \vdash_p \blacklozenge\ \tau\ x;\ t_1 \le x < t_2,\ p \ : \mathsf{int}}\,[\textsc{Quantif}]$$

Fig. 4. Type system inferring the types of terms and predicates for the generated code.

because of the C99 promotion rule [18, §6.3.1.1] that states (among others) that any expression more precise than `int` is automatically promoted to `int` when used in arithmetic operands. A left-value keeps its C type ([Lv]), while a logic binder takes it from the typing context ([Bind]). A cast $(\tau)t$ uses the interval inference system to downcast the resulting type of t to a possibly more precise type τ' (e.g. `char` for both `(int)4` and `(char)42`, cf. rule [Cast]). As usual with explicit coercions, it is up to the user to enforce safety. E-ACSL is also able to verify this safety property but that is outside the scope of this paper. The typing rule [Op] for an operator computes the kind of its operands and its result, merges them to get the most precise interval containing all of their possible values, and converts the result into the corresponding C type. The last two rules for terms are coercion rules. Rule [Sub] is a standard subsumption rule [19] (e.g., stating that any term of type `char` may have type `int` as well), while rule [$\downarrow$] soundly downcasts a term to a smaller type than its own type if its inferred kind fits in. For instance, the term `x/(y+1)` of Fig. 5 has type $\mathbb{Z} \hookleftarrow \mathbb{Z}$ after applying rule [Op], but since any possible result of the division fits into an `int` (i.e., $\theta(\kappa(x/(y+1))) \preccurlyeq_\tau$ `int`), it may soundly be coerced to this type. Yet its operands are still computed over $\mathbb{Z}$. Typing rules [Rel] and [Neg] for relations are straightforward. Rule [Cmd] for comparisons is similar to rule [Op], but the result is necessarily an `int` (actually either 0 or 1). A let-binding extends the typing context when evaluating the predicate ([Let]). A quantifier does the same ([Quantif]). In this latter case, the type associated to x is the smallest possible type with respect to its declared type and the one of its bounds. For instance, the inferred type of x is `char` for both $\forall$ `char` $x;\ -3 \le x < 4,\ 2{\times}x \le 6$ and $\forall$ `int` $x;\ 0 \le x < 4,\ 2{\times}x \le 6$ because x fits in $[0; 4]$ in each case. Fig. 5 shows the derivation for the latter formula. As for casts, it is up to the user to enforce safety of the type declaration, but this property may be verified independently.

Code generation. Generating code from the information computed by the type system is quite straightforward. Yet we may notice that the inference system is *not* an algorithm since several rules can be applied for a given term because of the coercion rules. A good implementation strategy consists in applying these coercions rules only when no other rules apply. This way, a cast must be introduced

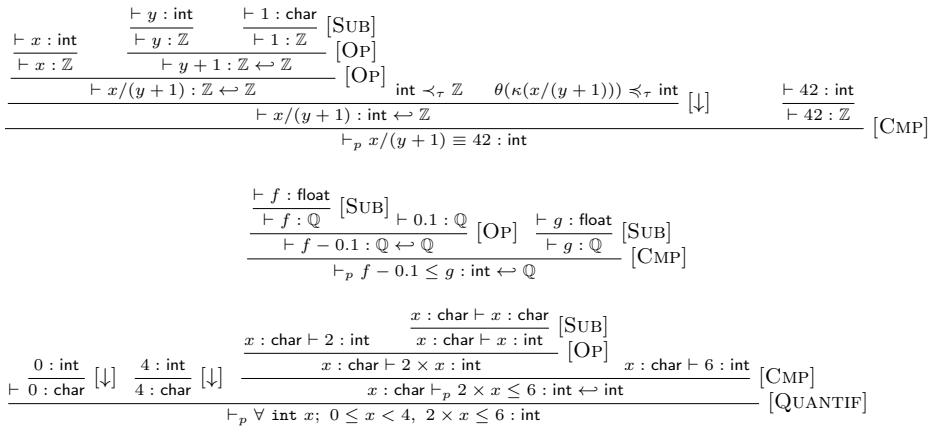

Fig. 5. Derivation trees for predicates $x/(y+1) \equiv 42$ (top), $f - 0.1 \leq g$ (middle) and $\forall\ \texttt{int}\ x;\ 0 \leq x < 4, 2 \times x \leq 6$ (bottom). The unlabeled rules are axioms or [Sub].

in the generated code if and only if a coercion rule is applied. Thus, this strategy introduces a minimal number of casts in the generated code. For instance, the code generated for the assertion `/*@ assert` $x/(y+1) \equiv 42 \wedge f - 0.1 \leq g$`; */` with the first operand of $\equiv$ of type `int` $\hookleftarrow \mathbb{Z}$, and f and g of type `double` would be as follows (the comments are manually added for readibility).

```
1  /* compute x/(y+1) with GMP integers */
2  mpz_t _x, _y, _cst_1, _add, _div; int _div2, _and;
3  mpz_init_set_si(_x, x); mpz_init_set_si(_y, y); mpz_init_set_si(_cst_1, 1);
4  mpz_init(_add); mpz_add(_add, _y, _cst_1);
5  mpz_init(_div); mpz_tdiv_q(_div, _x, _add);
6  /* soundly downcast the result of the division from GMP to int;
7     it corresponds to the application of rule [↓] in Fig 5. */
8  _div2 = mpz_get_si(_div);
9  if (_div2 == 42) {
10   /* compute f-0.1 ≤ g with GMP rationals */
11   mpq_t _f, _cst, _g, _sub; int _le;
12   mpq_init(_cst); mpq_set_str(_cst,"01/10",10);
13   mpq_init(_f); mpq_set_d(_f, f);
14   mpq_init(_sub); mpq_sub(_sub, _f, _cst);
15   mpq_init(_g); mpq_set_d(_g, g);
16   /* getting the result of the predicate as an int */
17   _le = mpq_cmp(_sub, _g);
18   _and = _le ≤ 0;
19   /* de-allocate the allocated GMP variables for rationals */
20   mpq_clear(_cst); mpq_clear(_f); mpq_clear(_sub); mpq_clear(_g);
21 } else
22 _and = 0;
23 /* runtime check the conjunction */
24 assert(_and);
25 /* de-allocate the allocated GMP variables for integers */
26 mpz_clear(_x); mpz_clear(_y); mpz_clear(_cst_1); mpz_clear(_add);
27 mpz_clear(_div);
```

It is worth noting that, at line 9, the code uses the C equality "`==`" to compare the result of the division to 42 thanks to our type system. The rational operations (lines 10–20) cannot be optimized because of the rational subtraction that cannot be soundly computed in any floating-point type, as we explained above. If the comparison were $f \leq g$, then the generated code would use a floating-point comparison. The effect of the type system is even clearer for the assertion

`/*@ assert` $c + 1 \equiv 0$`; */` in which the term $c + 1$ has type `int`: the generated code is as simple as the C assertion `assert(c+1 == 0);`

5 Experiments

We evaluate the benefits of the optimized code generation on several simple examples (cf. Fig. 6). The first one is a simple computational program similar to Fig. 1. It computes, given daily prices of a merchandise, its average price (as a floating-point value) over a year. The considered property states that the returned average has a maximal error ε with respect to the exact average. The other four examples are simple C programs with ACSL annotations involving integer numbers: three programs manipulating an array (binary search, search for a maximal element, search for repetitions), and a program dealing with a matrix (checking if a matrix is symmetric).

Since these examples are rather simple and their execution is very fast, we perform N runs (for various values of parameter N) of the computational part of each example, including annotations to be evaluated. This is done for two reasons. First, a unique execution is always instantaneous and thus the speedup computed for it is meaningless. Second, performing several iterations of the computational part simulates more complex examples where the evaluation of annotations represents a more significant part of the whole execution.

Figure 6 shows the execution time of the non-optimized (all-GMP) and optimized versions for different values of parameter N. It shows that the optimized code brings a speedup between 88% and 99% by replacing GMP code by machine-number based code. Moreover, some of the executions of the non-optimized version ran out-of-memory because of numerous heap allocations performed by the GMP code. Thanks to our optimization, this problem did not happen on the optimized version, where no GMP calls were required. Finally, the results of both versions were identical, giving confidence in the soundness of the optimization. Regarding the `average` example, it was executed with $\varepsilon = 0.001$. Our experiments also demonstrate that if the result type is `float`, the specified precision property is true for $\varepsilon = 0.001$, but fails for $\varepsilon = 0.0001$. If the computation result type is `double`, a much greater precision is achieved.

In general, the benefit of the proposed optimization for other programs can depend on the considered annotations and the amount of computation necessary to evaluate them with respect to the rest of the code. An evaluation of this optimization on real-life programs is left as future work, even if this optimization was already turned-on when benchmarking [20] or experimenting [21] E-ACSL.

These experiments also suggest that runtime assertion checking with the E-ACSL tool can be a very useful and easily performed step to empirically identify numerical properties (in particular, with respect to the exact result in mathematical numbers specified in ACSL) of a given code before attempting to perform their formal proof.

6 Conclusion and Future Work

We presented a type system parameterized by an interval analysis in order to rely on machine-number based code as often as possible when checking at runtime

		$N = 100$	$N = 1,000$	$N = 10,000$	$N = 100,000$	$N = 1,000,000$
average	All-GMP	0.008s	0.070s	0.719s	7.167s	73.817s
	Optimized	0.001s	0.006s	0.053s	0.567s	5.428s
	Speedup	88%	91%	93%	92%	93%
binary_search	All-GMP	0.041s	0.413s	4.625s	—	—
	Optimized	0.001s	0.005s	0.045s	0.474s	4.797s
	Speedup	98%	99%	99%		
max_seq	All-GMP	0.155s	1.620s	15.413s	—	—
	Optimized	0.003s	0.028s	0.278s	2.815s	29.793s
	Speedup	98%	98%	98%		
pair	All-GMP	0.142s	1.524s	15.813s	—	—
	Optimized	0.003s	0.026s	0.273s	2.603s	26.437s
	Speedup	98%	98%	98%		
symmetry	All-GMP	0.073s	0.758s	—	—	—
	Optimized	0.003s	0.021s	0.211s	2.106s	22.293s
	Speedup	96%	97%			

Fig. 6. Execution time of the intrumented examples with and without optimization, where "—" indicates that the execution exceeded the heap allocation limit of 128MB and thus the speedup cannot be computed.

properties over integer and rational numbers. It is implemented in the E-ACSL tool and has been adapted (for integers only) by Adacore to Spark2014. To the best of our knowledge, E-ACSL is the only runtime verification tool that is able to verify numerical properties over rational numbers. Our initial experiments confirm the soundness and the efficiency of our approach. More generally, it has already been used on large use cases a number of times without detecting soundness issues. Yet, evaluating the precise efficiency gain on larger use cases, as well as proving the soundness of the type system, are left as future work. Future work also includes more efficient generated code when dealing with $\mathbb{Q}$, and a richer support when dealing with numbers in $\mathbb{R}$.

Code generation. When dealing with rationals, arbitrary-precision code can be further reduced by using floating-point arithmetics when appropriate. Indeed, floating-point computations are exact under precisely defined circumstances. First, any multiplication and division by an integer power of two is exact: it simply corresponds to a change of the exponent in the binary representation of the floating-point number. Then, the Hauser theorem states that any floating-point addition is exact if both operands are small enough, while the Sterbenz lemma states that any floating-point subtraction is exact if the second operand is small enough [22, Theorem 3 and Lemma 2]. Several other floating-point number properties can be used in a similar way.

Expressiveness. The Richardson theorem states that equality over rational expressions extended with the sine function is undecidable [23]. Hence formal specifications that use the `sin` function, and any other trigonometric function by extension, cannot always be translated into *terminating* code. More generally, exact runtime assertion checking of properties over real numbers is not possible in finite time. To circumvent this issue, we could rely on sound approximations and partial verdicts (i.e., the tool would sometimes answer "I don't know").

Acknowledgment. The authors thank Thales Research & Technology for support of this work, the Frama-C team for providing the tool, as well as the anonymous reviewers for their helpful comments.

References

1. Clarke, L.A., Rosenblum, D.S.: A Historical Perspective on Runtime Assertion Checking in Software Development. SIGSOFT Software Engineering Notes **31**(3) (May 2006)
2. Kosmatov, N., Signoles, J.: A Lesson on Runtime Assertion Checking with Frama-C. In: International Conference on Runtime Verification (RV). (September 2013)
3. Chalin, P.: JML Support for Primitive Arbitrary Precision Numeric Types: Definition and Semantics. Journal of Object Technology **3**(6) (June 2004)
4. Chalin, P.: Improving JML: For a Safer and More Effective Language. In: International Symposium of Formal Methods Europe (FME). (September 2003)
5. Delahaye, M., Kosmatov, N., Signoles, J.: Common Specification Language for Static and Dynamic Analysis of C Programs. In: Symposium on Applied Computing (SAC). (March 2013)
6. Meyer, B.: Eiffel: The Language. Prentice-Hall (1992)
7. Barnett, M., Leino, K.R.M., Schulte, W.: The Spec# Programming System: An Overview. In: International Conference on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices (CASSIS). (March 2004)
8. Leavens, G.T., Baker, A.L., Ruby, C. In: JML: A Notation for Detailed Design. (October 1999)
9. Dross, C., Filliâtre, J.C., Moy, Y.: Correct Code Containing Containers. In: International Conference on Tests and Proofs (TAP). (June 2011)
10. Baudin, P., Filliâtre, J.C., Marché, C., Monate, B., Moy, Y., Prevosto, V.: ACSL: ANSI/ISO C Specification Language. http://frama-c.com/acsl.html.
11. Filliâtre, J.C., Paskevich, A.: Why3 — Where Programs Meet Provers. In: European Symposium on Programming (ESOP). (March 2013)
12. Kosmatov, N., Marché, C., Signoles, J., Moy, Y.: Static vs Dynamic Verification in Why3, Frama-C and SPARK 2014. In: International Symposium On Leveraging Applications of Formal Methods, Verification and Validation (ISoLA). (October 2016)
13. Signoles, J., Kosmatov, N., Vorobyov, K.: 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)
14. Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C: A Software Analysis Perspective. Formal Aspects of Computing (January 2015)
15. Signoles, J.: E-ACSL: Executable ANSI/ISO C Specification Language. http://frama-c.com/download/e-acsl/e-acsl.pdf.
16. Signoles, J.: From Static Analysis to Runtime Verification with Frama-C and E-ACSL (July 2018) Habilitation Thesis.
17. Jakobsson, A., Kosmatov, N., Signoles, J.: Rester statique pour devenir plus rapide, plus précis et plus mince. In: Journes Francophones des Langages Applicatifs (JFLA). (January 2015) In French.
18. ISO: ISO C Standard 1999. Technical report (1999)
19. Pierce, B.: Types and Programming Languages. MIT Press (2002)
20. Vorobyov, K., Signoles, J., Kosmatov, N.: Shadow state encoding for efficient monitoring of block-level properties. In: International Symposium on Memory Management (ISMM). (June 2017)
21. Pariente, D., Signoles, J.: 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)

22. Muller, J., Brisebarre, N., de Dinechin, F., Jeannerod, C., Lefèvre, V., Melquiond, G., Revol, N., Stehlé, D., Torres, S.: Handbook of Floating-Point Arithmetic. Birkhäuser (2010)
23. Richardson, D., Fitch, J.P.: The identity problem for elementary functions and constants. In: International Symposium on Symbolic and Algebraic Computation (ISSAC). (July 1994)