Symbolic transformation of expressions in modular arithmetic

Jérôme Boillot[1] and Jérôme Feret[1,2]

[1] École Normale Supérieure, Université PSL, Paris, France
[2] INRIA, Paris, France
`jerome.{boillot,feret}@ens.fr`

Abstract. We present symbolic methods to improve the precision of static analyses of modular integer expressions based on Abstract Interpretation. Like similar symbolic methods, the idea is to simplify on-the-fly arithmetic expressions before they are given to abstract transfer functions of underlying abstract domains. When manipulating fixed-length integer data types, casts and overflows generally act like modulo computations which hinder the use of symbolic techniques. The goal of this article is to formalize how modulo operations can be safely eliminated by abstracting arbitrary arithmetic expressions into sum, product, or division of linear forms with integer coefficients, while simplifying them. We provide some rules to simplify arithmetic expressions that are involved in the computation of linear interpolations, while ensuring the soundness of the transformation.
All these methods have been incorporated within the Astrée static analyzer that checks for the absence of run-time errors in embedded critical software, but also in an available toy abstract interpreter. The effects of our new abstract domain are then evaluated on several code excerpts from industrial code.

Keywords: Modular Arithmetic · Program Transformation · Symbolic Propagation · Abstract Intepretation · Interpolation.

SAS Artifact ★ Validated   SAS Artifact ★ Extensible   SAS Artifact ★ Available

1 Introduction

An important Computer Science challenge is to prove that given programs cannot crash. It is particularly needed in critical embedded applications such as planes, where potential errors can be fatal. Because of Rice's theorem, we know that it is impossible to create an analyzer that is both *automatic* (it does not require user interaction to finish, and can do so in finite time), *sound* (any proved

```
int x, y;
if (x >= y) {
  unsigned int r = (unsigned int) x - y;
  assert(r == (int64_t) x - y);
}
```

(a) C language

```
$X \leftarrow [-2^{31}, 2^{31}[$;
$Y \leftarrow [-2^{31}, 2^{31}[$;
if $X + -Y \geq 0$ then
  $R \leftarrow ((X$ mod $[0, 2^{32}[) + -(Y$ mod $[0, 2^{32}[))$
  $\hookrightarrow$   mod $[0, 2^{32}[$;
  // $R = X - Y$
endif
```

(b) Article's language

Fig. 1: Distance computation example.

property of the program actually holds), and *complete* (it is able to prove any property that holds). Because in general the most precise invariants are not computable, we decide to drop the completeness constraint. This means such analyzer can raise alarms that are false-positives. We are particularly interested in the Abstract Interpretation framework [5,6] and properties of integer expressions. Thus, the static analyzer we will present is parameterized by an auxiliary numerical abstract domain we will use to compute and represent numerical properties of program instructions.

There exist different numerical abstract domains that vary in precision, but also in time and memory costs. For example, we could name the interval domain [4] that computes sound variable bounds, and the polyhedra domain [8] that discovers linear inequalities. In order to prove the correctness of a program, it is sometimes necessary to retain information about the relationships between variables (*e.g.*, equalities between variables, or linear inequalities like in the polyhedron domain). Such abstract domains are called relational, and their usage can be very costly. That is why symbolic methods have been developed: to keep relations between variables by reasoning directly over the arithmetic expressions, and to apply sound program transformations on-the-fly to ease the analysis. This is what [21] describes with its abstraction of any arithmetic expressions into linear forms with interval coefficients. This abstraction allows algebraic simplifications.

When programming in languages, like C, that allow usage of fixed-length integers, it is important to consider how overflows and casts between different integer data types are handled. In C, this semantics is detailed in the C Standard [14]; we take into account that, in addition to explicit casts, some implicit casts are performed in arithmetic operations (*e.g.*, via the integer promotion). Unsigned integers do not overflow, the result is reduced modulo the largest value of the resulting type plus one. Casts also correspond to the use of a modulo. Thus, if we want to use symbolic methods, it appears necessary to deal with those modulo computations in the abstract representation.

In this paper, we present symbolic enhancement techniques similar to the ones described in [21], but that allow safe modulo elimination when it is possible. Consider, for instance, the program of Fig. 1 that computes the distance between X and Y when $X \geq Y$. As for every further introducing examples, we provide the C code in Fig. 1a and the representation in our internal language in Fig. 1b. Please note that in all the following examples the `int` data type is

```c
int x, p0, p1;
unsigned int a = (unsigned int) x - p0;
unsigned int b = (unsigned int) p1 - x;
if (p1 >= p0) {
  unsigned int r = a + b;
  assert(r == (int64_t) p1 - p0);
}
```

(a) C language

```
$X \leftarrow [-2^{31}, 2^{31}[$;
$P_0 \leftarrow [-2^{31}, 2^{31}[$;
$P_1 \leftarrow [-2^{31}, 2^{31}[$;
$A \leftarrow ((X \textbf{ mod } [0, 2^{32}[) + -(P_0 \textbf{ mod } [0, 2^{32}[))$
  ↪ $\textbf{mod } [0, 2^{32}[$;
$B \leftarrow ((P_1 \textbf{ mod } [0, 2^{32}[) + -(X \textbf{ mod } [0, 2^{32}[))$
  ↪ $\textbf{mod } [0, 2^{32}[$;
if $P_1 + -P_0 \geq 0$ then
  $R \leftarrow (A + B) \textbf{ mod } [0, 2^{32}[$;
  // $R = P_1 - P_0$
endif
```

(b) Article's language

Fig. 2: Variable elimination example.

```c
unsigned int x, x0, x1, y0, y1;
if (x0 <= x && x <= x1) {
  if (x0 != x1 && y0 <= y1) {
    unsigned int r = y0 + ((uint64_t)
  ↪  (x-x0) * (y1-y0) / (x1-x0));
    assert(y0 <= r && r <= y1);
  }
}
```

(a) C language

```
if $X + -X_0 \geq 0$ then
  if $X + -X_1 \leq 0$ then
    if $X_0 + -X_1 \neq 0$ then
      if $Y_0 + -Y_1 \leq 0$ then
        $R \leftarrow Y_0 + ((X + -X_0) \times (Y_1 + -Y_0)) /$
  ↪     $(X_1 + -X_0)$;
        // $Y_0 \leq R \leq Y_1$
      endif;
    endif;
  endif;
endif
```

(b) Article's language (simplified)

Fig. 3: First example of linear interpolation computation.

consistently represented using a 32-bit format. Nevertheless, this representation remains a parameter of the analysis. We expect the analyzer to infer that, when $X \geq Y$ is verified, $R = X - Y$. Note that $X \textbf{ mod } [0, 2^{32}[\neq X$ and the same holds with Y. So, traditional abstract domains like integers or polyhedron, or even linearization, could not infer this invariant. But our *modulo elimination* technique allows us to compute it. In our second example in Fig. 2 we would like to replace the expression $A+B$ by $P_1 - P_0$ by making the occurrences of the variable X in the expressions A and B cancel each other. This is made possible by the *symbolic constant propagation* domain. In addition, in our latest example in Fig. 3 (modulos are omitted for readability), if all conditions are met R is the linear interpolation of some function f in X such that $f(X_0) = Y_0$ and $f(X_1) = Y_1$. Then, R should be between Y_0 and Y_1. The abstract domain we present is able to compute such invariants thanks to the *interpolation detection step* in the reduction heuristic.

Related Work. The problem of analyzing programs with modular computations has already been addressed in the literature. Accurate results are especially important when inferring properties about pointer alignments and arrays lookup parallelization algorithms. The domains of congruences [11], linear congruences [12,22], trapezoidal congruences [18] have been used in that context. They offer several trade-offs for describing modular properties on intervals, linear inequalities, and rational linear inequalities. Modular arithmetic usually involves non-convex properties. A generic domain functor has been introduced in [23] to adapt abstract domains, so they can deal with modular properties.

The granularity of expression assignments is also important. For instance, while the single step assignment [9] method helps static analysis by decomposing the evaluation of expressions and by distinguishing multiple usages of each variable, it may also make more difficult symbolic simplifications of expressions. In contrast, symbolic constant propagation [21] allows composite assignments to be recombined to form composite expressions that can be simplified more easily. In the static analyzer Mopsa [16], abstract domains can rewrite expressions by resolving some aspects (such as pointers, floating point arithmetic), and simplify them symbolically. Lastly, some work has been done on fixed point arithmetic in the context of deductive methods [10].

Adapting symbolic simplification approaches in the context of modular arithmetic is different from detecting modular numerical properties. This is the issue we address in the present paper. Note that, while we focus on the simplification of expressions, the analysis of interpolation algorithms also requires precise handling of array lookup procedures. Analyzing array lookup loops and expressions involved in interpolation algorithms are orthogonal issues. The literature already describes some methods to address the former one [13,24].

In this work, we use an intermediate language inspired by the one used in [21]. We focus on integer arithmetic rather than floating-point computations, and we have introduced modulo computations and bound check operators. Moreover, we have adapted our rewriting relation from the one that is given in [21] by adding an explicit treatment of error alarms.

Outline. The paper is organized as follows. In Sect. 2, we present some preliminary results on modular integer arithmetic, which suggest it should be possible to effectively simplify expressions evaluated over modular rings. In Sect. 3, we introduce a toy arithmetic language that allows modulo computations and bound checks. The semantics of this language describes two kinds of error alarms: divisions by zero and failed bound checks. In Sect. 4 we explain how the integration of the expression rewriting technique can be done soundly. Then, in Sect. 5 we introduce an abstract representation of expressions that can be tuned by the parameters of a generic numerical abstract domain described in Sect. 6. Such a generic abstract domain is then instantiated in Sect. 7, by making explicit the heuristics used to symbolically simplify expressions. Finally, Sect. 8 describes some aspects of our implementations and an evaluation of the introduced abstract domain over several code excerpts from industrial code.

2 Preliminary Results on Modular Integer Arithmetic

In this section, we give basic properties to reason on Euclidean division and modular arithmetic. First, we introduce the set of modulo specifications $\mathbb{M}$ that is defined as follows:

$$\mathbb{M} \overset{\text{def}}{=} \{\, [l, u[\mid l, u \in \mathbb{Z},\ l < u \,\} \cup \{\, \mathbb{Z} \,\}.$$

The set $\mathbb{M}$ contains two kinds of elements. Intervals of the form $[l, u[$ denote operations modulo $u - l$ with result in the interval $[l, u[$, whereas the element $\mathbb{Z}$ denotes arithmetic operations without modulo. This is formalized in the following definition:

Definition 1. *We define the modulo of $k \in \mathbb{Z}$ by an element of $\mathbb{M}$ as*

$$\begin{aligned} k \bmod [l, u[&\overset{\text{def}}{=} l + \operatorname{irem}(k - l, u - l) \\ k \bmod \mathbb{Z} &\overset{\text{def}}{=} k \end{aligned}$$

where $\operatorname{irem}(m, n)$ denotes the remainder of the Euclidean division of the integer m by the strictly positive integer n, that is to say the unique integer r such that $0 \le r < n$ and n divides $m - r$.

In particular, when the integer k belongs to the interval $[l, u[$ we have $k \bmod [l, u[= k$.

Definition 2. *An element $m \in \mathbb{M}$ is said to be k-splittable, with k a strictly positive integer, if m can be split in k sets of same cardinality. We define $\mathbb{S} : \mathbb{N}^* \to \wp(\mathbb{M})$ such that $\mathbb{S}(k)$ is the set of elements of $\mathbb{M}$ that are k-splittable. More formally,*

$$\forall k > 0,\ \mathbb{S}(k) \overset{\text{def}}{=} \{\, [l, u[\in \mathbb{M} \mid (k \mid (u - l)) \,\} \cup \{\, \mathbb{Z} \,\}.$$

Example 1. The intervals $[0, 2^{16}[$ and $[-2^{15}, 2^{15}[$ both are 2^{16}-splittable and 2^8-splittable.

We use the notion of k-splittability to reason about the application of consecutive modulo computations. In particular, when two modulo operations follow each other, the inner one can be ignored under some specific conditions. This is formalized in the following property.

Property 1. Let $n \in \mathbb{Z}$ be an integer. Let $[l, u[$ and m be two elements of $\mathbb{M}$. If m is $(u - l)$-splittable (*i.e.*, if $m \in \mathbb{S}(u - l)$), then $(n \bmod m) \bmod [l, u[= n \bmod [l, u[$.

We now give two other properties to simplify consecutive modulo computations under some conditions.

Property 2. Let $n \in \mathbb{Z}$ be an integer. Let $[l_1, u_1[$ and $[l_2, u_2[$ be two elements of $\mathbb{M}$ such that $[l_1, u_1[\subseteq [l_2, u_2[$. Then, $(n \bmod [l_1, u_1[) \bmod [l_2, u_2[= n \bmod [l_1, u_1[$.

Property 3. Let $n \in \mathbb{Z}$ be an integer. Let $[l_1, u_1[$ and $[l_2, u_2[$ be two elements of $\mathbb{M}$. We consider $\alpha \overset{\text{def}}{=} l_1 \bmod [l_2, u_2[$. If $\alpha + u_1 - l_1 \le u_2$ and $(u_1 - l_1) \mid (\alpha - l_1)$, then we have $(n \bmod [l_1, u_1[) \bmod [l_2, u_2[= n \bmod [\alpha, \alpha + u_1 - l_1[$. This is pictured in Fig. 4.

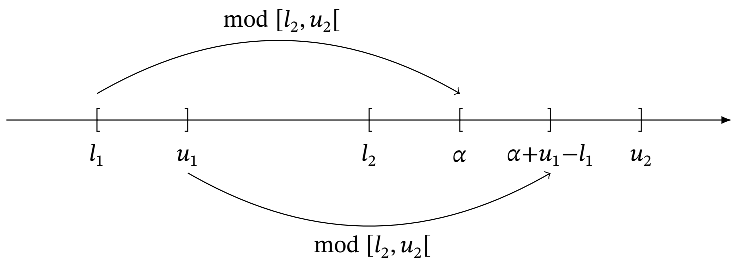


Fig. 4: Translation when applying consecutive modulos.

$expr ::=$	X	$X \in \mathcal{V}$
$\mid$	$[a, b]$	$a, b \in \mathbb{Z},\ a \leq b$
$\mid$	$-expr$	
$\mid$	$expr \diamond expr$	$\diamond \in \{+, \times, /\}$
$\mid$	$expr \uplus expr$	
$\mid$	$expr$ **mod** $[l, u[$	$l, u \in \mathbb{Z}, l < u$
$\mid$	**bound_check**$(expr, [l, u[)$	$l, u \in \mathbb{Z}, l < u$
$stmt ::=$	$X \leftarrow expr$	$X \in \mathcal{V}$
$\mid$	**if** $expr \bowtie 0$ **then** $stmt$; **endif**	$\bowtie \in \{=, \neq, <, \leq, \geq, >\}$
$\mid$	**while** $expr \bowtie 0$ **do** $stmt$; **done**	$\bowtie \in \{=, \neq, <, \leq, \geq, >\}$
$\mid$	$stmt$; $stmt$	

Fig. 5: Syntax of our extended language.

Note that Prop. 2 is a particular case of Prop. 3. Indeed, with the notations of Prop. 3, if $[l_1, u_1[\subseteq [l_2, u_2[$ we have $\alpha = l_1$. Then, $\alpha + u_1 - l_1 = u_1$ and $\alpha - l_1 = 0$. So, $\alpha + u_1 - l_1 \leq u_2$ and $(u_1 - l_1) \mid (\alpha - l_1)$. We can conclude, thanks to Prop. 3, that $(n \text{ mod } [l_1, u_1[) \text{ mod } [l_2, u_2[= n \text{ mod } [l_1, u_1[$

We finish this preliminary results section, with few examples about modulo computations.

Example 2. Let n be an integer in $\mathbb{Z}$. Then,
- by Prop. 1, $(n \text{ mod } [-2^{15}, 2^{15}[) \text{ mod } [0, 2^8[= n \text{ mod } [0, 2^8[$,
- because $23 \in [0, 2^8[$, $23 \text{ mod } [0, 2^8[= 23$,
- by Prop. 2, $(n \text{ mod } [0, 2^8[) \text{ mod } [0, 2^{16}[= n \text{ mod } [0, 2^8[$,
- by Prop. 3, $(n \text{ mod } [2, 4[) \text{ mod } [10, 20[= n \text{ mod } [12, 14[$ (with $\alpha = 12$).

3 Syntax and Semantics of the Language

3.1 Syntax of the Language

The syntax of our language is introduced in Fig. 5. The analysis that is described in this paper only refines the result of integer computations. Thus, the description of pointers and floating point arithmetic is omitted. We indeed suppose that lvalue resolution has been partially solved (see [2, Sect. 6.1.3]) and that the abstraction presented here assigns no information to the value of floating point variables (which are handled by some other abstract domains of the analyzer). We focus on integer expressions, which are made of variables, constant intervals, classical arithmetic operations, modulo within a constant modular ring, and bound checks. A specific operator $\uplus$ is added to our extended syntax to represent the value of some sub-expressions. This operator represents the convex join of two integer expressions, that is any value between its operands. This is especially helpful to deal with expressions involved in interpolation procedures. For example, in Fig. 3, if it is defined, the value of R is between the values of Y_0 and Y_1, regardless of their order. We then denote $R \leftarrow Y_0 \uplus Y_1$. Assuming we can establish that $Y_0 \leq Y_1$, it follows that $Y_0 \leq R \leq Y_1$.

The variables involved in the program belong to a finite set, denoted $\mathcal{V}$. We use interval constants to represent constants or to model non-determinism that may be due to some unknown inputs or potentially imprecise abstraction. Implicit and explicit casts have been decomposed by means of two operators: as it will be seen in the description of the semantics, the bound check operator **bound_check** checks whether the value of an expression does not overflow, and the modulo operator **mod** extracts the remainder of the Euclidean division. Lastly, we assume that bit shifting has been replaced with equivalent arithmetic operations.

Statements include assignments, sequential composition, conditional branching (we only consider positive branches, negative ones can be encoded consequently), and loops. The conditions of branching and loops compare expressions with the value 0. This toy language is enough to encode the semantics of the integer arithmetic restriction of real-life programming languages like C.

For the rest of the paper, every constant expression of the form $[\alpha, \alpha]$ with $\alpha \in \mathbb{Z}$ is denoted α_{cst}.

3.2 Concrete Semantics of the Language

We now describe the *concrete semantics* of our language, that is a mathematical expression of its behaviors. It is worth noting that, due to the use of intervals and convex join operators, the evaluation of an expression may induce non-determinism. Additionally, our semantics tracks *erroneous computations*.

We introduce a set of possible errors that we note Ω. We assume that Ω contains in particular two distinct elements ω_d and ω_o: the error ω_d stands for a division by zero, whereas the error ω_o denotes a bound check failure. The other elements of Ω can be raised by the other domains of the analyzer, especially during the resolution of the lvalues.

A *memory state* is a function that maps each variable from the set $\mathcal{V}$ to an integer in $\mathbb{Z}$. The set of all memory states is denoted $\mathcal{E}$. The concrete semantics $[\![\mathit{expr}]\!] \in \mathcal{E} \to (\wp(\mathbb{Z}) \times \wp(\Omega))$ of an expression maps a memory state to sets of values and errors. To simplify the formulas, given an expression $e \in \mathit{expr}$ and a memory state $\rho \in \mathcal{E}$, the first component of $[\![e]\!]\rho$ is often written $[\![e]\!]^V \rho \in \wp(\mathbb{Z})$,

$$
\begin{aligned}
&[\![X]\!]\rho \stackrel{\text{def}}{=} \langle \{ \rho(X) \},\ \varnothing \rangle \\
&[\![[a,b]]\!]\rho \stackrel{\text{def}}{=} \langle \{ x \in \mathbb{Z} \mid a \leq x \leq b \},\ \varnothing \rangle \\
&[\![-e]\!]\rho \stackrel{\text{def}}{=} \langle \{ -x \mid x \in [\![e]\!]^V \rho \},\ [\![e]\!]^\Omega \rho \rangle \\
&[\![e_1 + e_2]\!]\rho \stackrel{\text{def}}{=} \langle \{ x + y \mid x \in [\![e_1]\!]^V \rho, y \in [\![e_2]\!]^V \rho \},\ [\![e_1]\!]^\Omega \rho \cup [\![e_2]\!]^\Omega \rho \rangle \\
&[\![e_1 \times e_2]\!]\rho \stackrel{\text{def}}{=} \langle \{ x \times y \mid x \in [\![e_1]\!]^V \rho, y \in [\![e_2]\!]^V \rho \},\ [\![e_1]\!]^\Omega \rho \cup [\![e_2]\!]^\Omega \rho \rangle \\
&[\![e_1 \,/\, e_2]\!]\rho \stackrel{\text{def}}{=} \langle \{ \mathrm{truncate}(x \,/\, y) \mid x \in [\![e_1]\!]^V \rho, y \in [\![e_2]\!]^V \rho, y \neq 0 \},\ [\![e_1]\!]^\Omega \rho \cup [\![e_2]\!]^\Omega \rho \cup \Omega_1 \rangle \\
&\qquad\qquad\qquad\qquad \text{with } \Omega_1 \stackrel{\text{def}}{=} \begin{cases} \{ \omega_d \} & \text{if } 0 \in [\![e_2]\!]^V \rho \\ \varnothing & \text{otherwise} \end{cases} \\
&[\![e_1 \Cup e_2]\!]\rho \stackrel{\text{def}}{=} \left\langle \left\{ z \in \mathbb{Z} \;\middle|\; \begin{array}{l} x \in [\![e_1]\!]^V \rho,\ y \in [\![e_2]\!]^V \rho \\ x \leq z \leq y \ \vee\ y \leq z \leq x \end{array} \right\},\ [\![e_1]\!]^\Omega \rho \cup [\![e_2]\!]^\Omega \rho \right\rangle \\
&[\![e \ \mathbf{mod}\ [l,u[\ ]\!]\rho \stackrel{\text{def}}{=} \langle \{ x \bmod [l,u[\ \mid x \in [\![e]\!]^V \rho \},\ [\![e]\!]^\Omega \rho \rangle \\
&[\![\mathbf{bound\_check}(e, [l,u[)]\!]\rho \stackrel{\text{def}}{=} \langle [\![e]\!]^V \rho,\ [\![e]\!]^\Omega \rho \cup \Omega_1 \rangle \text{ with } \Omega_1 \stackrel{\text{def}}{=} \begin{cases} \{ \omega_o \} & \text{if } [\![e]\!]^V \rho \not\subseteq [l,u[\\ \varnothing & \text{otherwise} \end{cases}
\end{aligned}
$$

Fig. 6: Concrete semantics of expressions.

while the second one is written $[\![e]\!]^\Omega \rho \in \wp(\Omega)$. These notations are used in the inductive definition given in Fig. 6 and also until the rest of the paper.

The evaluation of a variable raises no error, it only gives the value that is fetched from the memory state. The evaluation of an interval constant raises no error either. It provides an arbitrary value in the corresponding interval. Classical arithmetic operators and modulo consist in applying the corresponding operations element-wise while propagating the errors potentially raised when evaluating their sub-expressions. Additionally, the result of every division between two integers is *truncated*, that is rounded towards zero, while raising the error ω_d when the denominator can take the value 0. The convex join operator $\Cup$ outputs any value between the potential values of its operands and propagates the errors, but does not raise new ones. Finally, bound-checking propagates values while raising the error ω_o when they do not fit in the interval given as an argument.

It is worth to note that division by 0 and overflows are handled differently: executions that perform divisions by 0 are halted (they produce no memory states), whereas those that cause overflows are continued without modifying the current value. It corresponds to user-defined semantics assumptions [1], which are more specific than C standard which assumes the result is non-deterministic.

We now define the concrete semantics of statements. The semantics domain $\mathcal{D}$ collects both sets of memory states and errors. Thus, it is defined as follows:

$$\mathcal{D} \stackrel{\text{def}}{=} \wp(\mathcal{E}) \times \wp(\Omega).$$

It is equipped with the component-wise join $\sqcup$ and order $\sqsubseteq$.

The concrete semantics of a statement, $\{\!| \mathit{stmt} |\!\} : \mathcal{D} \to \mathcal{D}$, maps each element of the semantics domain $\rho \in \mathcal{D}$ to another one. For a given statement, it applies

$$\{\!|X \leftarrow e|\!\}\langle R_0, \Omega_0\rangle \stackrel{\text{def}}{=} \langle \emptyset, \Omega_0\rangle \sqcup \bigsqcup_{\rho \in R_0} \left\{ \langle \{\rho[X \mapsto v]\}, [\![e]\!]^{\Omega}\rho\rangle \,\middle|\, v \in [\![e]\!]^{V}\rho \right\}$$

$$\{\!|e \bowtie 0?|\!\}\langle R_0, \Omega_0\rangle \stackrel{\text{def}}{=} \langle \emptyset, \Omega_0\rangle \ \sqcup \bigsqcup_{\rho \in R_0} \left\{ \langle \{\rho\}, [\![e]\!]^{\Omega}\rho\rangle \,\middle|\, \exists v \in [\![e]\!]^{V}\rho,\ v \bowtie 0 \right\}$$

$$\{\!|s_1 \, ; s_2|\!\} \stackrel{\text{def}}{=} \{\!|s_2|\!\} \circ \{\!|s_1|\!\}$$

$$\{\!|\,\textbf{if}\ e \bowtie 0\ \textbf{then}\ s;\ \textbf{endif}\,|\!\}\langle R_0, \Omega_0\rangle \stackrel{\text{def}}{=} (\{\!|s|\!\} \circ \{\!|e \bowtie 0?|\!\})\langle R_0, \Omega_0\rangle \ \sqcup\ \{\!|e \not\bowtie 0?|\!\}\langle R_0, \Omega_0\rangle$$

$$\{\!|\,\textbf{while}\ e \bowtie 0\ \textbf{do}\ s;\ \textbf{done}\,|\!\}\langle R_0, \Omega_0\rangle \stackrel{\text{def}}{=} \{\!|e \not\bowtie 0?|\!\}\Big(\bigsqcup_{n \in \mathbb{N}} (\{\!|s|\!\} \circ \{\!|e \bowtie 0?|\!\})^{n} \langle R_0, \Omega_0\rangle \Big)$$

Fig. 7: Concrete semantics of statements.

the transformation over the possible memory states and accumulates the potential errors. It is defined by induction over the syntax in Fig. 7. The operator $\bigsqcup$ refers to the iteration of the binary associative and commutative operator $\sqcup$ over the elements of the set given as an argument.

Roughly speaking, the set of potential memory states, after assigning an expression $e \in expr$ to a variable $X \in \mathcal{V}$, is obtained by considering each potential memory state before the execution of the assignment and each potential value for the expression in that memory state ; for each combination, the memory state is updated by taking into account the potential value of the expression. $\rho[X \leftarrow x]$ denotes the function equal to ρ on $\mathcal{V} \setminus \{X\}$ and that maps X to x. The evaluation of the expression can also yield errors, which are also collected. The semantics of the sequential composition of two statements is the composition of their semantics. Lastly, the semantics of conditional branching and loops rely on the handling of guarding conditions: the execution of the guard restricts the set of memory states to those that satisfy the corresponding condition. The potential errors raised when evaluating the expression are also collected. The semantics of conditional branching apply the semantics of the true branch on the result of the application of the guard, and join it to the result of the application of the negation of the guard. Lastly, the semantics of loops is obtained by unfolding the loop according to its number of iterations.

4 Soundness requirements of expression rewriting

We now introduce a rewriting order over expressions, noted $\preccurlyeq$, and parameterized by a set of error alarms. Rewriting an expression may rely on some conditions about the current state of the system. Additionally, it can simplify some parts of the initial expression, which could potentially raise some error alarms. The set of these error alarms are reported as side-conditions.

Definition 3. *The relation $\preccurlyeq_{\Omega_1}$, with $\Omega_1 \in \wp(\Omega)$ a set of potential error alarms, is defined as follows:*

$$\langle R_0, \Omega_0\rangle \vDash e_1 \preccurlyeq_{\Omega_1} e_2 \ \stackrel{\text{def}}{=}\ \forall \rho \in R_0,\ [\![e_1]\!]^{V}\rho \subseteq [\![e_2]\!]^{V}\rho \ \wedge\ [\![e_1]\!]^{\Omega}\rho \subseteq [\![e_2]\!]^{\Omega}\rho \cup \Omega_1$$

with $\langle R_0, \Omega_0 \rangle \in \mathcal{D}$ *a semantics element and* $e_1, e_2 \in expr$ *two expressions.*

The definition of the rewriting relation $\preccurlyeq_{\Omega_1}$ is based on the semantics of expressions. Given a semantics element $\langle R_0, \Omega_0 \rangle \in \mathcal{D}$, an expression e_1 can be rewritten in the expression e_2 if and only if, in every memory state $\rho \in R_0$, the potential values of the expression e_1 are all potential values of the expression e_2. Yet, simplifications of the expression e_1 may hide error alarms, which are reported in the set Ω_1.

Example 3. We wonder whether the expression $0_{cst} \times \textbf{bound\_check}(X, [0, 2^8[)$ can be simplified into the expression 0_{cst}. It depends on the potential range of the variable X. Let R_0 be a set of memory states. If, for every memory state $\rho \in R_0$, we have $\rho(X) \in [0, 2^8[$, then $\langle R_0, \varnothing \rangle \models 0_{cst} \times \textbf{bound\_check}(X, [0, 2^8[) \preccurlyeq_{\varnothing} 0_{cst}$. Otherwise, we can prove that $\langle R_0, \varnothing \rangle \models 0_{cst} \times \textbf{bound\_check}(X, [0, 2^8[) \preccurlyeq_{\{\omega_o\}} 0_{cst}$. We can check that the rewriting order warns about the potential failure of the bound check, despite the fact that the expression that contains this bound check has been removed by simplifications.

Example 4. The expression $((X \textbf{ mod } [0, 2^8[) + -(Y \textbf{ mod } [0, 2^8[)) \textbf{ mod } [0, 2^8[$ can be rewritten as $X - Y$ under some specific assumptions. Let R_0 be a set of memory states. If for every memory state $\rho \in R_0$, we have $\rho(X), \rho(Y) \in [-2^7, 2^7[$ and $\rho(X) - \rho(Y) \in [0, 2^8[$, it follows that $\langle R_0, \varnothing \rangle \models ((X \textbf{ mod } [0, 2^8[) + -(Y \textbf{ mod } [0, 2^8[)) \textbf{ mod } [0, 2^8[\preccurlyeq_{\varnothing} X + -Y$. Indeed, either no cast wraps-around, or exactly two among the three that appear in the expression. In the later case, they compensate each other.

The following property states the transitivity of the rewriting relation.

Property 4. For every semantics element $\langle R_0, \Omega_0 \rangle \in \mathcal{D}$, expressions $e_1, e_2, e_3 \in$ *expr*, and sets of error alarms $\Omega_1, \Omega_1' \in \wp(\Omega)$, if $\langle R_0, \Omega_0 \rangle \models e_1 \preccurlyeq_{\Omega_1} e_2$ and $\langle R_0, \Omega_0 \rangle \models e_2 \preccurlyeq_{\Omega_1'} e_3$, then $\langle R_0, \Omega_0 \rangle \models e_1 \preccurlyeq_{\Omega_1 \cup \Omega_1'} e_3$.

Transitivity is obtained by evaluating the expression with the same memory states and by collecting all the error alarms that may be hidden by the expression rewritings.

In statements, expressions can be rewritten. The following theorem states the soundness of the replacement of an expression by another one in assignments or guards. It would also be true for every statement of the language, yet we omit this result since it is not necessary to prove the soundness of our analysis.

Theorem 1. *For every semantics element* $\langle R_0, \Omega_0 \rangle \in \mathcal{D}$*, expressions* $e, e' \in$ *expr, set of error alarms* $\Omega_1 \in \wp(\Omega)$*, and variable* $X \in \mathcal{V}$ *such that* $\langle R_0, \Omega_0 \rangle \models e \preccurlyeq_{\Omega_1} e'$*, we have:*

- $\llbracket V \leftarrow e \rrbracket \langle R_0, \Omega_0 \rangle \sqsubseteq \langle \varnothing, \Omega_1 \rangle \sqcup (\llbracket V \leftarrow e' \rrbracket \langle R_0, \Omega_0 \rangle)$,
- $\llbracket e \bowtie 0? \rrbracket \langle R_0, \Omega_0 \rangle \sqsubseteq \langle \varnothing, \Omega_1 \rangle \sqcup (\llbracket e' \bowtie 0? \rrbracket \langle R_0, \Omega_0 \rangle)$.

This way, when an expression is replaced by another one in an assignment or a guard, while following the rewriting order, all the possible memory states and error alarms are kept in the result. Note that the approximation may lead to the introduction of additional memory states or false-negative error alarms.

$$\begin{array}{rcl} \mathit{expr}^\sharp & ::= & \boxed{\mathcal{L}}\left(a_0 + \sum_{X_i \in \mathcal{V}} a_i X_i\right) \quad \forall i, a_i \in \mathbb{Z} \\ & \mid & \mathit{expr}^\sharp \boxplus \mathit{expr}^\sharp \\ & \mid & \mathit{expr}^\sharp \boxtimes \mathit{expr}^\sharp \\ & \mid & \mathit{expr}^\sharp \boxslash \mathit{expr}^\sharp \\ & \mid & \mathit{expr}^\sharp \boxed{\Cup} \mathit{expr}^\sharp \end{array}$$

Fig. 8: Syntax of abstract expressions.

5 Abstract Representation of Expressions

As far as it is possible to totally order their variables, *linear expressions* have a canonical representation. It is obtained by factorizing occurrences of each variable and ordering their terms increasingly, with respect to the order on the variables. We introduce in this section an abstract syntax for expressions, in which some linear expressions are described canonically. The main goal is to highlight the patterns that can be simplified symbolically.

5.1 Abstract Syntax of Expression

The abstract syntax of expressions is given in Fig. 8. Apart from linear combinations, abstract expressions are defined using the same operators as in the language syntax. For the sake of rigor, and to distinguish them from their concrete counterparts, all the abstract operators are enclosed within a box.

Linear combinations are written as $\boxed{\mathcal{L}}\left(a_0 + \sum_{X_i \in \mathcal{V}} a_i X_i\right)$ with $a_0 \in \mathbb{Z}$ and $\forall i, a_i \in \mathbb{Z}$. In particular, we use the convention that every variable in the set $\mathcal{V}$ has to occur in the expression, would it be with a zero coefficient. This eases the definition of operations over linear forms. Constants are specific linear combinations, where all coefficients, except potentially the first one, are equal to 0. We write $\boxed{\mathcal{L}}(\alpha)$ the constant whose first coefficient is equal to $\alpha \in \mathbb{Z}$. The set of all such abstract expressions is denoted $\mathit{Const}^\sharp$. A variable is a linear combination of which all the coefficients are fixed at 0, except that of the variable, which is equal to 1. It is denoted $\boxed{\mathcal{L}}(X)$. Lastly, variable differences are linear combinations of which all the coefficients are fixed to 0, except for two variables. One of them has the coefficient 1 and the other -1. The variable difference between X and Y, two variables of $\mathcal{V}$, is written $\boxed{\mathcal{L}}(X - Y)$.

Intervals are introduced by the means of the convex join operator $\boxed{\Cup}$. Lastly, bound checks and modulo computations are not described. Indeed, bound checks are assumed to have been eliminated while reporting the potential error alarms. About modulo computations, abstract expressions are given with an evaluation context. This context takes the form of a modular ring specified by an element of $\mathbb{M}$. Inner modulo computations are assumed to have been resolved, either by proving that they leave the value of the expression unchanged, or by replacing them conservatively by an interval.

The meaning of an abstract expression is defined thanks to a function toExpr which translates abstract expressions in the set $expr^\sharp$ back to expressions in the set $expr$.

Definition 4. *The function* toExpr : $expr^\sharp \to expr$ *is defined inductively as follows:*

$$\begin{aligned}
&\mathrm{toExpr}\left(\boxed{\mathcal{L}}\left(a_0 + \textstyle\sum_{X_i \in \mathcal{V}} a_i X_i\right)\right) \overset{\text{def}}{=} {a_0}_{cst} + ({a_1}_{cst} X_1 + (\cdots + ({a_n}_{cst} X_n)))\\
&\mathrm{toExpr}(e_1^\sharp \boxplus e_2^\sharp) \overset{\text{def}}{=} \mathrm{toExpr}(e_1^\sharp) + \mathrm{toExpr}(e_2^\sharp)\\
&\mathrm{toExpr}(e_1^\sharp \boxtimes e_2^\sharp) \overset{\text{def}}{=} \mathrm{toExpr}(e_1^\sharp) \times \mathrm{toExpr}(e_2^\sharp)\\
&\mathrm{toExpr}(e_1^\sharp \boxslash e_2^\sharp) \overset{\text{def}}{=} \mathrm{toExpr}(e_1^\sharp) \,/\, \mathrm{toExpr}(e_2^\sharp)\\
&\mathrm{toExpr}(e_1^\sharp \boxed{\Cup}\, e_2^\sharp) \overset{\text{def}}{=} \mathrm{toExpr}(e_1^\sharp) \Cup \mathrm{toExpr}(e_2^\sharp)
\end{aligned}$$

6 Generic Abstraction

The translation of classical expressions into abstract ones is parametric, with respect to the choice of an abstract domain, to reason about semantics states. The abstract domain describes a set of properties about semantics states $\mathcal{D}^\sharp$, which are mapped to the least upper bound of the semantics states which satisfy them by a concretization function γ. It also contains a sound abstract counterpart to the join operator $\sqcup$ noted $\sqcup^\sharp$, a primitive $\mathrm{lfp}^\sharp$ to approximate the increasing iteration of concrete operators, and two abstract transformers $\textsc{assign}^\sharp$ and $\textsc{guard}^\sharp$ to lift the execution of assignments and guards on properties. It also contains the primitives ι and **reduce**. The primitive ι extracts the range of abstract expressions and **reduce** performs sound expression rewriting for all the memory states contained in the semantics state given as an argument.

Definition 5. *An abstract domain consists of a tuple comprising eight elements* $\langle\, \mathcal{D}^\sharp, \gamma, \sqcup^\sharp, \mathrm{lfp}^\sharp, \textit{ASSIGN}^\sharp, \textit{GUARD}^\sharp, \iota, \textbf{\textit{reduce}} \,\rangle$*, such that:*

5.1 $\mathcal{D}^\sharp$ *is a set of properties,*

5.2 $\gamma : \mathcal{D}^\sharp \to \mathcal{D}$ *is a concretization function that, given an abstract element* $R^\sharp$*, outputs all the memory states and error alarms that verify the property* $R^\sharp$*,*

5.3 *for every two abstract elements* $R^\sharp, S^\sharp \in \mathcal{D}^\sharp$*,* $\gamma(R^\sharp) \sqcup \gamma(S^\sharp) \sqsubseteq \gamma(R^\sharp \sqcup^\sharp S^\sharp)$*,*

5.4 *for every abstract element* $R^\sharp \in \mathcal{D}^\sharp$ *and every abstract transformer* $\mathbb{F}^\sharp : \mathcal{D}^\sharp \to \mathcal{D}^\sharp$*,* $lfp^\sharp_{R^\sharp}(\mathbb{F}^\sharp)$ *is an abstract element that satisfies* $\bigsqcup_{n\in\mathbb{N}} \mathbb{F}^n(\langle\, R_0, \Omega_0 \,\rangle) \sqsubseteq \gamma(lfp^\sharp_{R^\sharp}(\mathbb{F}^\sharp))$ *for every semantics element* $\langle\, R_0, \Omega_0 \,\rangle \in \mathcal{D}$ *and every* $\sqcup$*-complete morphism*[3] $\mathbb{F} : \mathcal{D} \to \mathcal{D}$ *such that:*
   - *(i)* $\langle\, R_0, \Omega_0 \,\rangle \sqsubseteq \mathbb{F}(\langle\, R_0, \Omega_0 \,\rangle)$*,*
   - *(ii)* $\langle\, R_0, \Omega_0 \,\rangle \sqsubseteq \gamma(R^\sharp)$*,*

[3] that is to say $\mathbb{F}\left(\bigsqcup P\right) = \bigsqcup\{\, \mathbb{F}(\langle\, R_0, \Omega_0 \,\rangle) \mid \langle\, R_0, \Omega_0 \,\rangle \in P \,\}$ for every set $P \subseteq \mathcal{D}$.

$$\{\!| X \leftarrow e |\!\}^\sharp \overset{\text{def}}{=} \text{ASSIGN}^\sharp(X, E)$$
$$\{\!| s_1 ; s_2 |\!\}^\sharp \overset{\text{def}}{=} \{\!| s_2 |\!\}^\sharp \circ \{\!| s_1 |\!\}^\sharp$$
$$\{\!| \textbf{if } e \bowtie 0 \textbf{ then } s; \textbf{ endif} |\!\}^\sharp R^\sharp \overset{\text{def}}{=} ((\{\!| s |\!\}^\sharp \circ \text{GUARD}^\sharp(e, \bowtie))R^\sharp) \sqcup \text{GUARD}^\sharp(e, \not\bowtie)R^\sharp$$
$$\{\!| \textbf{while } e \bowtie 0 \textbf{ do } s; \textbf{ done} |\!\}^\sharp R^\sharp \overset{\text{def}}{=} \text{GUARD}^\sharp(e, \not\bowtie)(\text{lfp}^\sharp [X^\sharp \mapsto R^\sharp \sqcup^\sharp (\{\!| s |\!\}^\sharp \circ \text{GUARD}^\sharp(e, \bowtie))X^\sharp])$$

Fig. 9: Abstract semantics.

(iii) for every abstract element $S^\sharp \in \mathcal{D}^\sharp$, $(\mathbb{F} \circ \gamma)(S^\sharp) \sqsubseteq (\gamma \circ \mathbb{F}^\sharp)(S^\sharp)$,

5.5 *for every variable* $X \in \mathcal{V}$, *and every expression* $e \in$ *expr*, $\textit{ASSIGN}^\sharp(X, e) : D^\sharp \to D^\sharp$ *is a function that satisfies* $(\{\!| X \leftarrow e |\!\} \circ \gamma)R^\sharp \sqsubseteq (\gamma \circ \textit{ASSIGN}^\sharp(X, e))R^\sharp$ *for every abstract element* $R^\sharp \in \mathcal{D}^\sharp$,

5.6 *for every comparison relation* $\bowtie \in \{=, \neq, <, \leq, \geq, >\}$, *and every expression* $e \in$ *expr*, $\textit{GUARD}^\sharp(e, \bowtie) : D^\sharp \to D^\sharp$ *is a function that satisfies* $(\{\!| e \bowtie 0? |\!\} \circ \gamma)R^\sharp \sqsubseteq (\gamma \circ \textit{GUARD}^\sharp(e, \bowtie))R^\sharp$ *for every abstract element* $R^\sharp \in \mathcal{D}^\sharp$,

5.7 *for every expression* $e \in$ *expr*, $\iota(e) : \mathcal{D}^\sharp \to \mathbb{I}$ *is a function that satisfies* $[\![e]\!]^V \rho \subseteq \iota(e)R^\sharp$ *for every abstract element* $R^\sharp \in \mathcal{D}^\sharp$ *and for every memory state* $\rho \in R_0$ *with* $\langle R_0, \Omega_0 \rangle = \gamma(R^\sharp)$ *and* $\mathbb{I} \overset{\text{def}}{=} \{\emptyset\} \cup \{[a, b] \mid a \in \{-\infty\} \cup \mathbb{Z}, b \in Z \cup \{+\infty\}, a \leq b\}$, *the set of intervals over* $\mathbb{Z} \cup \{-\infty, +\infty\}$,

5.8 *for every abstract expression* $e^\sharp$, $\boldsymbol{reduce}(e^\sharp) : \mathcal{D}^\sharp \to expr^\sharp$ *is an abstract expression transformer such that the rewriting relation* $\gamma(R^\sharp) \models \text{toExpr}(e^\sharp) \preccurlyeq_\emptyset \text{toExpr}(\boldsymbol{reduce}(e^\sharp)R^\sharp)$ *holds for every abstract element* $R^\sharp$.

The $\text{lfp}^\sharp$ operator is usually described as an increasing iteration, followed by a decreasing iteration. These are defined by the means of a base abstract element, a widening operator, and a narrowing operator [7]. We now assume that such an abstract domain has been chosen.

The abstract semantics of a statement $\{\!| stmt |\!\}^\sharp : \mathcal{D}^\sharp \to \mathcal{D}^\sharp$ maps a property about semantics states, before applying the statement *stmt*, to the property that is satisfied after applying this statement. It is obtained by lifting the concrete semantics (*e.g.* see Fig. 7) in the abstract domain. Its definition is given in Fig. 9. Each concrete operation is replaced with its abstract counterpart. The abstraction of loops requires more explanations. In order to apply the abstract operator $\text{lfp}^\sharp$, we must ensure that the first argument is a sound approximation of a monotonic function, and that the second argument is an abstraction of a pre-fixpoint of this function. Hence, we replace the function $(\{\!| s |\!\} \circ \{\!| e \bowtie 0? |\!\})$ from the iteration, in the concrete semantics, by the function $\langle R_0, \Omega_0 \rangle \mapsto \langle R_0, \Omega_0 \rangle \sqcup (\{\!| s |\!\} \circ \{\!| e \bowtie 0? |\!\})\langle R_0, \Omega_0 \rangle$, where $\langle R_0, \Omega_0 \rangle$ is the semantics state just before interpreting the loop. This does not change the result of the concrete iterations.

We can now state the soundness of the abstract semantics.

Theorem 2. *Let $\langle R_0, \Omega_0 \rangle \in \mathcal{D}$ be a semantics state. Let $R^\sharp \in \mathcal{D}^\sharp$ be an abstract state. Let $s \in stmt$ be a statement. Then $\langle R_0, \Omega_0 \rangle \sqsubseteq \gamma(R^\sharp) \implies \llbracket s \rrbracket \langle R_0, \Omega_0 \rangle \sqsubseteq \gamma(\llbracket s \rrbracket^\sharp R^\sharp)$.*

Thm. 2 states that the abstract semantics ignores no concrete behavior. Nevertheless, it may introduce fictitious ones due to the abstraction.

6.1 Primitives over abstract expressions

We now introduce two primitives that operate over abstract expressions.

The opposite function pushes unary minus to the leafs of abstract expressions (that are linear forms).

Definition 6. *The function* opposite : *$expr^\sharp \rightarrow expr^\sharp$ is defined inductively as follows:*

$$\begin{aligned}
&\mathsf{opposite}\left(\boxed{\mathbb{Z}}\left(a_0 + \textstyle\sum_{X_i \in \mathcal{V}} a_i X_i\right)\right) \overset{\text{def}}{=} \boxed{\mathbb{Z}}\left(-a_0 + \textstyle\sum_{X_i \in \mathcal{V}} (-a_i) X_i\right)\\
&\mathsf{opposite}(e_1^\sharp \boxplus e_2^\sharp) \overset{\text{def}}{=} \mathsf{opposite}(e_1^\sharp) \boxplus \mathsf{opposite}(e_2^\sharp)\\
&\mathsf{opposite}(e_1^\sharp \boxtimes e_2^\sharp) \overset{\text{def}}{=} \mathsf{opposite}(e_1^\sharp) \boxtimes e_2^\sharp\\
&\mathsf{opposite}(e_1^\sharp \boxslash e_2^\sharp) \overset{\text{def}}{=} \mathsf{opposite}(e_1^\sharp) \boxslash e_2^\sharp\\
&\mathsf{opposite}(e_1^\sharp \boxed{\uplus} e_2^\sharp) \overset{\text{def}}{=} \mathsf{opposite}(e_1^\sharp) \boxed{\uplus} \mathsf{opposite}(e_2^\sharp)
\end{aligned}$$

This way, the opposite of a linear form is obtained by taking the opposite of each coefficient. The function opposite propagates over the sub-expressions of the $\boxplus$ and the $\boxed{\uplus}$ operators. Lastly, the opposite of a product or a quotient between two sub-expressions is obtained by propagating it to only one of them (the first one has been chosen arbitrarily).

We now introduce an operator to propagate a modulo computation over an abstract expression. Given an abstract expression and a modulo specification, it applies the modulo on the expression.

Definition 7. *For any abstract expression $e^\sharp \in expr^\sharp$ and any modulo specification $m \in \mathbb{M}$, the function* $\mathrm{rmMod}(e, m) : \mathcal{D}^\sharp \rightarrow expr^\sharp$ *is defined as follows:*

$$\mathrm{rmMod}(e^\sharp, m)R^\sharp \overset{\text{def}}{=} \begin{cases}
e^\sharp & \textit{if } m = \mathbb{Z},\\
\boxed{\mathbb{Z}}(\alpha \bmod m) & \textit{else if } e^\sharp = \boxed{\mathbb{Z}}(\alpha) \textit{ with } \alpha \in \mathbb{Z},\\
e^\sharp & \textit{else if } \iota(\mathrm{toExpr}(e^\sharp))R^\sharp \subseteq m,\\
\boldsymbol{reduce}(\boxed{\mathbb{Z}}(l) \boxed{\uplus} \boxed{\mathbb{Z}}(u-1))R^\sharp & \textit{otherwise, with } m = [l, u[.
\end{cases}$$

In the previous definition, if the modulo specification is equal to the set $\mathbb{Z}$, nothing has to be done. When the abstract expression is a constant, the modulo computation can be directly applied on the constant. Lastly, if the value of the abstract expression ranges within the interval of the modulo, then the modulo computation can be safely discarded. In all other cases, no information can be kept about the expression. It is then replaced with the interval of the modular ring (or more precisely its reduction).

The following theorem states that any abstract expression that is evaluated over a modular ring can be rewritten in the expression in which the modulo computation has been forced, that is to say the output of the function rmMod. Moreover, this rewrite does not hide any potential error alarms.

Theorem 3. *For all abstract value $R^\sharp \in \mathcal{D}^\sharp$, abstract expression $e^\sharp \in expr^\sharp$, and potential modular ring $m \in \mathbb{M}$, the following property holds,*

$$\gamma(R^\sharp) \vDash \text{toExpr}(e^\sharp)\ \mathit{mod}\ m \preccurlyeq_\varnothing \text{toExpr}(\text{rmMod}(e^\sharp, m)).$$

Example 5. We give two examples of elimination of modulo computations. We compute the result of the abstract expression $\boxed{\mathcal{L}}(0)\ \boxed{\Cup}\ \boxed{\mathcal{L}}(25)$, that intuitively denotes the interval $[0, 25]$, respectively modulo $[0, 2^8[$ and $[10, 26[$. We assume that the primitive ι provides the exact range of this expression, that is to say that $\iota(\text{toExpr}(\boxed{\mathcal{L}}(0)\ \boxed{\Cup}\ \boxed{\mathcal{L}}(25)))R^\sharp = \iota(0_{cst} \Cup 25_{cst}) = [0, 25]$. Since the interval $[0, 25]$ is included in the interval $[0, 2^8[$, the corresponding modulo computation can be eliminated without modifying the abstract expression. We obtain $\text{rmMod}(\boxed{\mathcal{L}}(0)\ \boxed{\Cup}\ \boxed{\mathcal{L}}(25), [0, 2^8[)R^\sharp = \boxed{\mathcal{L}}(0)\ \boxed{\Cup}\ \boxed{\mathcal{L}}(25)$. In the second case, the result of the modulo computation cannot be described precisely as an abstract expression. It is then soundly replaced by the abstract expression $\mathbf{reduce}(\boxed{\mathcal{L}}(10)\ \boxed{\Cup}\ \boxed{\mathcal{L}}(25))R^\sharp$. We keep the primitive **reduce** uninterpreted, since it is a parameter of our abstraction.

6.2 Translation from Classical to Abstract Expressions

We now have all the material needed to define the translation of classical expressions into abstract ones. Given an expression $e \in expr$, its translation $(\!|\, e \,|\!) : \mathcal{D}^\sharp \to expr^\sharp \times \mathbb{M} \times \wp(\Omega)$ is a function that maps an abstract element $R^\sharp$ to a triple $(e^\sharp, m, \Omega_e)$. Remember that abstract expressions do not have modulo operators. However, the element $m \in \mathbb{M}$ stands for a modulo computation to be applied on the potential values of the abstract expression. This way, the outermost modulo computation can be kept precisely, whereas inner modulo computations must be translated conservatively. This can be done thanks to the rmMod operator, yet it may yield a loss of information. Bound checks also cannot be described in abstract expressions, so potential bound check failures are reported in the set Ω_e. The translation is only valid for the semantics states satisfying the property $R^\sharp$ (*i.e.*, for the states in $\gamma(R^\sharp)$). Thus, the abstract element $R^\sharp$ should be used to drive the translation process to get a more accurate result.

The translation $(\!|\, e \,|\!)R^\sharp$ of an expression e, in the context of an abstract element $R^\sharp$, is defined inductively and by cases by the means of a set of inference rules.

Variable. A variable is replaced by a linear combination where all the coefficients are set to 0, except the one corresponding to the variable which is set to 1.

Such a replacement hides no error alarms, and the abstract expression obtained this way can be interpreted in $\mathbb{Z}$. This is formalized in the following inference rule.

$$\textsc{Variable}\ \frac{X \in \mathcal{V}}{(\!|\, X \,|\!)R^\sharp \overset{\text{def}}{=} (\boxed{\mathcal{L}}(X), \mathbb{Z}, \varnothing)}$$

Interval. An interval is encoded by the means of the convex join operator $\boxed{\Cup}$. The bounds of the interval are given as operands (their order has been chosen arbitrarily). Then, the **reduce** operator is applied to potentially simplify the resulting abstract expression. This translation yields no potential errors and its result can be interpreted in $\mathbb{Z}$. This is formalized in the following inference rule.

$$\textsc{Interval}\ \frac{a, b \in \mathbb{Z} \qquad a \leq b}{(\!|\, [a,b] \,|\!)R^\sharp \overset{\text{def}}{=} (\mathbf{reduce}(\boxed{\mathcal{L}}(a) \boxed{\Cup} \boxed{\mathcal{L}}(b))R^\sharp, \mathbb{Z}, \varnothing)}$$

Unary minus. The translation of an expression starting with a unary minus is defined inductively. First, the argument is translated, which provides an abstract expression, a modulo specification, and a set of potential errors. The primitive opposite is then applied to the abstract expression, which yields no additional error alarms. Furthermore, when the abstract expression is evaluated over a modular ring, the ring is kept the same, but the elements of the potential modular interval are also negated.

This is formalized in the two following inference rules, which distinguish two cases according to whether the translation of the argument can be interpreted in $\mathbb{Z}$, or in a modular ring.

$$\textsc{OppositeNoMod}\ \frac{(e^\sharp, \mathbb{Z}, \Omega_e) \overset{\text{def}}{=} (\!|\, e \,|\!)R^\sharp}{(\!|\, -e \,|\!)R^\sharp \overset{\text{def}}{=} (\mathbf{reduce}(\mathrm{opposite}(e^\sharp))R^\sharp, \mathbb{Z}, \Omega_e)}$$

$$\textsc{OppositeMod}\ \frac{(e^\sharp, [l, u[, \Omega_e) \overset{\text{def}}{=} (\!|\, e \,|\!)R^\sharp}{(\!|\, -e \,|\!)R^\sharp \overset{\text{def}}{=} (\mathbf{reduce}(\mathrm{opposite}(e^\sharp))R^\sharp, [-u+1, -l+1[, \Omega_e)}$$

Convex join. An expression of the form $e_1 \Cup e_2$ is translated thanks to its abstract counterpart $\boxed{\Cup}$. First, the sub-expressions are translated, which provides abstract expressions, modulo specifications, and sets of potential errors. The outermost modulo computations are conservatively suppressed using the rmMod primitive before the results are passed to the $\boxed{\Cup}$ abstract operator. The final result may be simplified by the means of the **reduce** operator before being interpreted in $\mathbb{Z}$. No additional potential errors are collected. This is formalized in the following inference rule.

$$\textsc{ConvexJoin}\ \frac{\begin{array}{cc} (e_1^\sharp, m_1, \Omega_1) \overset{\text{def}}{=} (\!|\, e_1 \,|\!)R^\sharp & (e_2^\sharp, m_2, \Omega_2) \overset{\text{def}}{=} (\!|\, e_2 \,|\!)R^\sharp \\ e_1'^\sharp \overset{\text{def}}{=} \mathrm{rmMod}(e_1^\sharp, m_1)R^\sharp & e_2'^\sharp \overset{\text{def}}{=} \mathrm{rmMod}(e_2^\sharp, m_2)R^\sharp \end{array}}{(\!|\, e_1 \Cup e_2 \,|\!)R^\sharp \overset{\text{def}}{=} (\mathbf{reduce}(e_1'^\sharp \boxed{\Cup} e_2'^\sharp)R^\sharp, \mathbb{Z}, \Omega_1 \cup \Omega_2)}$$

Addition. According to the result of the translation of its arguments, more or less precise inference rules can be used to translate a sum of two expressions.

Whenever both operands are translated into constants, whether they need to be respectively interpreted in modular rings or not, the potential application of the modulo operations can be directly applied on the constant values. The result can be interpreted in $\mathbb{Z}$. Whenever exactly one operand is translated into a constant, and the other one must be interpreted in modular arithmetic, the potential modulo operator of the constant expression can be directly applied. Then, the result is added to the other abstract expression and to the bounds of its modular ring. The resulting abstract expression may be simplified by the means of the **reduce** operator. In the context of real programming languages, branching is generally not limited to comparisons to 0. Then, we would like to take advantage of the algebraic simplifications of our abstract domain and rewrite `e1 == e2` into `e1 - e2 == 0`. We then need a rule that simplifies addition of abstract expressions interpreted in the same modular ring that sum to $\boxed{\mathbb{Z}}(0)$. In such a case, the result is $\boxed{\mathbb{Z}}(0)$ and can be interpreted in $\mathbb{Z}$. Otherwise, the operator rmMod is used to remove modulo operators in both translations of the arguments. It yields abstract expressions which can be interpreted in $\mathbb{Z}$, but may result in loss of information. The resulting abstract expression can be potentially simplified by the means of the **reduce** parametric operator.

In all cases, the computation yields no additional error alarms. This is formalized in the following four inference rules (we recall that by convention $\alpha \ \mathsf{mod}\ \mathbb{Z}$ is equal to α for every integer $\alpha \in \mathbb{Z}$).

$$\textsc{Plus2Const}\ \frac{\begin{array}{c}\alpha_1 \in \mathbb{Z} : (\boxed{\mathbb{Z}}(\alpha_1), m_1, \Omega_1) \stackrel{\text{def}}{=} \llparenthesis\, e_1 \,\rrparenthesis R^\sharp \\ \alpha_2 \in \mathbb{Z} : (\boxed{\mathbb{Z}}(\alpha_2), m_2, \Omega_2) \stackrel{\text{def}}{=} \llparenthesis\, e_2 \,\rrparenthesis R^\sharp\end{array}}{\llparenthesis\, e_1 + e_2 \,\rrparenthesis R^\sharp \stackrel{\text{def}}{=} (\boxed{\mathbb{Z}}(\alpha_1 \bmod m_1 + \alpha_2 \bmod m_2), \mathbb{Z}, \Omega_i \cup \Omega_j)}$$

$$\textsc{PlusConst}\ \frac{\begin{array}{c}i, j \in \{1,2\} : i \neq j \qquad \alpha \in \mathbb{Z} : (\boxed{\mathbb{Z}}(\alpha), m_i, \Omega_i) \stackrel{\text{def}}{=} \llparenthesis\, e_i \,\rrparenthesis R^\sharp \\ (e_j^\sharp, [l_j, u_j[, \Omega_j) \stackrel{\text{def}}{=} \llparenthesis\, e_j \,\rrparenthesis R^\sharp \qquad \alpha' \stackrel{\text{def}}{=} \alpha \bmod m_i \qquad e_j^\sharp \notin \mathit{Const}^\sharp\end{array}}{\llparenthesis\, e_1 + e_2 \,\rrparenthesis R^\sharp \stackrel{\text{def}}{=} (\mathbf{reduce}(\boxed{\mathbb{Z}}(\alpha') \boxplus e_j^\sharp)R^\sharp, [l_j{+}\alpha', u_j{+}\alpha'[, \Omega_i \cup \Omega_j)}$$

$$\textsc{PlusEqZero}\ \frac{\begin{array}{c}(e_1^\sharp, [l_1, u_1[, \Omega_1) \stackrel{\text{def}}{=} \llparenthesis\, e_1 \,\rrparenthesis R^\sharp \\ (e_2^\sharp, [l_2, u_2[, \Omega_2) \stackrel{\text{def}}{=} \llparenthesis\, e_2 \,\rrparenthesis R^\sharp \qquad \mathbf{reduce}(e_1^\sharp \boxplus e_2^\sharp)R^\sharp = \boxed{\mathbb{Z}}(0) \\ l_2 = -u_1 + 1 \qquad u_2 = -l_1 + 1 \qquad e_1^\sharp \notin \mathit{Const}^\sharp \qquad e_2^\sharp \notin \mathit{Const}^\sharp\end{array}}{\llparenthesis\, e_1 + e_2 \,\rrparenthesis R^\sharp \stackrel{\text{def}}{=} (\boxed{\mathbb{Z}}(0), \mathbb{Z}, \Omega_1 \cup \Omega_2)}$$

$$\textsc{PlusNoMod}\ \frac{\begin{array}{c}(e_1^\sharp, m_1, \Omega_1) \stackrel{\text{def}}{=} \llparenthesis\, e_1 \,\rrparenthesis R^\sharp \qquad (e_2^\sharp, m_2, \Omega_2) \stackrel{\text{def}}{=} \llparenthesis\, e_2 \,\rrparenthesis R^\sharp \\ e_1'^\sharp \stackrel{\text{def}}{=} \mathrm{rmMod}(e_1^\sharp, m_1)R^\sharp \qquad e_2'^\sharp \stackrel{\text{def}}{=} \mathrm{rmMod}(e_2^\sharp, m_2)R^\sharp \\ \text{neither the rule } \textsc{Plus2Const} \text{ nor } \textsc{PlusConst} \text{ nor } \textsc{PlusEqZero} \text{ can be applied}\end{array}}{\llparenthesis\, e_1 + e_2 \,\rrparenthesis R^\sharp \stackrel{\text{def}}{=} (\mathbf{reduce}(e_1'^\sharp \boxplus e_2'^\sharp)R^\sharp, \mathbb{Z}, \Omega_1 \cup \Omega_2)}$$

Note that in case one argument is translated into a constant and the other one into an abstract expression interpreted in $\mathbb{Z}$, then applying the rmMod operator produces no loss of information. Thus, the rule PlusNoMod is enough.

Multiplication. The translation of a multiplication between two expressions works similarly.

In the case both operands are translated into constants, the potential application of the modulo operation can be directly applied on the constant values. The result can then be interpreted in $\mathbb{Z}$. Whenever exactly one operand is translated into a constant and the other abstract expression must be interpreted in modular arithmetic, the potential modulo operator of the constant expression can be directly applied on the constant value. The resulting abstract expression is multiplied by the constant and then potentially simplified by the **reduce** operator. The update of the modular interval depends on the sign of the constant, which splits the inference rule into three ones, depending on whether the constant is positive, zero, or negative. In all other cases, the rmMod operator is used to suppress the modulo computations in both arguments translations. Once again, the resulting abstract expression can be potentially simplified by the means of the **reduce** parametric operator.

In all cases, the computation yields no additional error alarms. This is formalized in the following five inference rules.

$$\textsc{Mult2Const}\ \frac{\begin{array}{c}\alpha_1 \in \mathbb{Z} : (\boxed{\mathcal{C}}(\alpha_1), m_1, \Omega_1) \stackrel{\text{def}}{=} \llparenthesis\, e_1 \,\rrparenthesis R^\sharp \\ \alpha_2 \in \mathbb{Z} : (\boxed{\mathcal{C}}(\alpha_2), m_2, \Omega_2) \stackrel{\text{def}}{=} \llparenthesis\, e_2 \,\rrparenthesis R^\sharp\end{array}}{\llparenthesis\, e_1 \times e_2 \,\rrparenthesis R^\sharp \stackrel{\text{def}}{=} (\boxed{\mathcal{C}}((\alpha_1 \bmod m_1) \times (\alpha_2 \bmod m_2)), \mathbb{Z}, \Omega_i \cup \Omega_j)}$$

$$\textsc{MultPosConst}\ \frac{\begin{array}{c} i, j \in \{1,2\} : i \neq j \\ \alpha \in \mathbb{Z} : (\boxed{\mathcal{C}}(\alpha), m_i, \Omega_i) \stackrel{\text{def}}{=} \llparenthesis\, e_i \,\rrparenthesis R^\sharp \qquad (e_j^\sharp, [l_j, u_j[, \Omega_j) \stackrel{\text{def}}{=} \llparenthesis\, e_j \,\rrparenthesis R^\sharp \\ \alpha' \stackrel{\text{def}}{=} \alpha \bmod m_i \qquad \alpha' > 0 \qquad e_j^\sharp \notin \mathit{Const}^\sharp \end{array}}{\llparenthesis\, e_1 \times e_2 \,\rrparenthesis R^\sharp \stackrel{\text{def}}{=} (\mathbf{reduce}(e_j^\sharp \boxtimes \boxed{\mathcal{C}}(\alpha'))R^\sharp, [l_j{\times}\alpha', u_j{\times}\alpha'[, \Omega_i \cup \Omega_j)}$$

$$\textsc{MultZeroConst}\ \frac{\begin{array}{c} i, j \in \{1,2\} : i \neq j \\ \alpha \in \mathbb{Z} : (\boxed{\mathcal{C}}(\alpha), m_i, \Omega_i) \stackrel{\text{def}}{=} \llparenthesis\, e_i \,\rrparenthesis R^\sharp \qquad (e_j^\sharp, m_j, \Omega_j) \stackrel{\text{def}}{=} \llparenthesis\, e_j \,\rrparenthesis R^\sharp \\ \alpha' \stackrel{\text{def}}{=} \alpha \bmod m_i \qquad \alpha' = 0 \qquad e_j^\sharp \notin \mathit{Const}^\sharp \end{array}}{\llparenthesis\, e_1 \times e_2 \,\rrparenthesis R^\sharp \stackrel{\text{def}}{=} (\boxed{\mathcal{C}}(0), \mathbb{Z}, \Omega_i \cup \Omega_j)}$$

$$\textsc{MultNegConst}\ \frac{\begin{array}{c} i, j \in \{1,2\} : i \neq j \\ \alpha \in \mathbb{Z} : (\boxed{\mathcal{C}}(\alpha), m_i, \Omega_i) \stackrel{\text{def}}{=} \llparenthesis\, e_i \,\rrparenthesis R^\sharp \qquad (e_j^\sharp, [l_j, u_j[, \Omega_j) \stackrel{\text{def}}{=} \llparenthesis\, e_j \,\rrparenthesis R^\sharp \\ \alpha' \stackrel{\text{def}}{=} \alpha \bmod m_i \qquad \alpha' < 0 \qquad e_j^\sharp \notin \mathit{Const}^\sharp \end{array}}{\llparenthesis\, e_1 \times e_2 \,\rrparenthesis R^\sharp \stackrel{\text{def}}{=} (\mathbf{reduce}(e_j^\sharp \boxtimes \boxed{\mathcal{C}}(\alpha'))R^\sharp, [u_j\alpha'{+}1, l_j\alpha'{+}1[, \Omega_i \cup \Omega_j)}$$

$$\textsc{MultNoMod}\ \frac{\begin{array}{c} (e_1^\sharp, m_1, \Omega_1) \stackrel{\text{def}}{=} \llparenthesis\, e_1 \,\rrparenthesis R^\sharp \qquad (e_2^\sharp, m_2, \Omega_2) \stackrel{\text{def}}{=} \llparenthesis\, e_2 \,\rrparenthesis R^\sharp \\ e_1'^\sharp \stackrel{\text{def}}{=} \mathrm{rmMod}(e_1^\sharp, m_1)R^\sharp \qquad e_2'^\sharp \stackrel{\text{def}}{=} \mathrm{rmMod}(e_2^\sharp, m_2)R^\sharp \\ \text{neither the rule } \textsc{Mult2Const} \text{ nor } \textsc{MultPosConst} \\ \text{nor } \textsc{MultZeroConst} \text{ nor } \textsc{MultNegConst} \text{ can be applied} \end{array}}{\llparenthesis\, e_1 \times e_2 \,\rrparenthesis R^\sharp \stackrel{\text{def}}{=} (\mathbf{reduce}(e_1'^\sharp \boxtimes e_2'^\sharp)R^\sharp, \mathbb{Z}, \Omega_1 \cup \Omega_2)}$$

Division. Propagating modular computations across divisions is quite tricky. Indeed, it can be done precisely only when the following conditions are met. First, the numerator has to be positive. We then consider the modulo interval of the numerator. The denominator has to be a nonzero constant that divides both bounds of this interval (or the bounds of the interval that contains the opposite values when the constant is negative). In addition, this interval must not include both negative (*i.e.*, < 0) and positive (*i.e.*, > 0) values. That is why two inference rules are provided, depending on the sign of both the denominator and the elements of the modulo interval. In such a case, the abstract expression is obtained by reducing the result of the application of the $\boxslash$ operator to the translation of its operands, thanks to the **reduce** primitive. This expression can be evaluated in modular arithmetic: the resulting modular ring is obtained by dividing by the constant both bounds of the modular ring of the numerator (or its opposite when the constant is negative). Because the constant is not 0, the computation does not yield additional error alarms. In all other cases, the rmMod operator is used to suppress the modulo computations in both translations of its arguments, and its result can be directly interpreted in $\mathbb{Z}$, at the cost of a possible loss of information. The resulting abstract expression can be potentially simplified by the means of the **reduce** parametric operator. Such computation also has to collect the potential error alarm ω_d when the primitive ι is unable to prove that the value of the denominator cannot be 0.

This is formalized in the following three inference rules.

$$\textsc{DivPosConst}\ \dfrac{\begin{array}{c}\alpha' \in \mathbb{Z} : (e_1^\sharp, [\alpha' l_1, \alpha' u_1[, \Omega_1) \stackrel{\text{def}}{=} \llparenthesis\, e_1 \,\rrparenthesis R^\sharp \\ \alpha \in \mathbb{Z} : (\boxed{\mathbb{Z}}(\alpha), m_2, \Omega_2) \stackrel{\text{def}}{=} \llparenthesis\, e_2 \,\rrparenthesis R^\sharp \qquad \alpha' \stackrel{\text{def}}{=} \alpha \bmod m_2 \\ \alpha' > 0 \qquad \iota(\text{toExpr}(e_1^\sharp))R^\sharp \subseteq [0, +\infty[\qquad l_1 \geq 0\end{array}}{\llparenthesis\, e_1 \,/\, e_2 \,\rrparenthesis R^\sharp \stackrel{\text{def}}{=} (\mathbf{reduce}(e_1^\sharp \boxslash \boxed{\mathbb{Z}}(\alpha'))R^\sharp, [l_1, u_1[, \Omega_1 \cup \Omega_2)}$$

$$\textsc{DivNegConst}\ \dfrac{\begin{array}{c}\alpha' \in \mathbb{Z} : (e_1^\sharp, [\alpha'(u_1-1), \alpha'(l_1-1)[, \Omega_1) \stackrel{\text{def}}{=} \llparenthesis\, e_1 \,\rrparenthesis R^\sharp \\ \alpha \in \mathbb{Z} : (\boxed{\mathbb{Z}}(\alpha), m_2, \Omega_2) \stackrel{\text{def}}{=} \llparenthesis\, e_2 \,\rrparenthesis R^\sharp \qquad \alpha' \stackrel{\text{def}}{=} \alpha \bmod m_2 \\ \alpha' < 0 \qquad \iota(\text{toExpr}(e_1^\sharp))R^\sharp \subseteq [0, +\infty[\qquad u_1 \leq 1\end{array}}{\llparenthesis\, e_1 \,/\, e_2 \,\rrparenthesis R^\sharp \stackrel{\text{def}}{=} (\mathbf{reduce}(e_1^\sharp \boxslash \boxed{\mathbb{Z}}(\alpha'))R^\sharp, [l_1, u_1[, \Omega_1 \cup \Omega_2)}$$

$$\textsc{DivNoMod}\ \dfrac{\begin{array}{c}(e_1^\sharp, m_1, \Omega_1) \stackrel{\text{def}}{=} \llparenthesis\, e_1 \,\rrparenthesis R^\sharp \\ (e_2^\sharp, m_2, \Omega_2) \stackrel{\text{def}}{=} \llparenthesis\, e_2 \,\rrparenthesis R^\sharp \qquad e_1'^\sharp \stackrel{\text{def}}{=} \text{rmMod}(e_1', m_1)R^\sharp \\ e_2'^\sharp \stackrel{\text{def}}{=} \text{rmMod}(e_2', m_2)R^\sharp \qquad \Omega_3 \stackrel{\text{def}}{=} \begin{cases}\{\,\omega_d\,\} & \text{if } 0 \in \iota(\text{toExpr}(e_2'^\sharp))R^\sharp \\ \varnothing & \text{otherwise}\end{cases} \\ \text{neither the rule } \textsc{DivPosConst} \text{ nor } \textsc{DivNegConst} \text{ can be applied}\end{array}}{\llparenthesis\, e_1 \,/\, e_2 \,\rrparenthesis R^\sharp \stackrel{\text{def}}{=} (\mathbf{reduce}(e_1'^\sharp \boxslash e_2'^\sharp)R^\sharp, \mathbb{Z}, \Omega_1 \cup \Omega_2 \cup \Omega_3)}$$

Bound check. Bound check expressions may warn about potential overflows and underflows. First, the expression in the argument of the bound check is translated. Then, if its potential values can be proven to be necessarily within the bounds checked, no additional alarm has to be collected. Otherwise, a potential error ω_o is collected.

Different methods can be used to compute the range of possible values of the translated expression. Whenever the interval of the outermost modulo of the inner expression is included in the interval of the bound check, there is no additional alarm to record. Otherwise, the primitive rmMod is used to eliminate the outermost modulo applied to the abstract expression. Then, the ι primitive is used to collect the range of the result. This range is checked against the bounds of the bound check.

This is formalized in the following two inference rules.

$$\textsc{BoundCheckMod}\ \frac{(e^\sharp, m, \Omega_e) \stackrel{\text{def}}{=} (\!(e)\!)R^\sharp \qquad m \subseteq [l, u[}{(\!(\textbf{bound\_check}(e, [l, u[))\!)R^\sharp \stackrel{\text{def}}{=} (e^\sharp, m, \Omega_e)}$$

$$\textsc{BoundCheckNoMod}\ \frac{\begin{array}{c}(e^\sharp, m, \Omega_e) \stackrel{\text{def}}{=} (\!(e)\!)R^\sharp \qquad m \not\subseteq [l, u[\\ \Omega_1 \stackrel{\text{def}}{=} \begin{cases}\{\omega_o\} & \text{if } \iota(\text{toExpr}(\text{rmMod}(e^\sharp, m)R^\sharp))R^\sharp \not\subseteq [l, u[\\ \varnothing & \text{otherwise}\end{cases}\end{array}}{(\!(\textbf{bound\_check}(e, [l, u[))\!)R^\sharp \stackrel{\text{def}}{=} (e^\sharp, m, \Omega_e \cup \Omega_1)}$$

Modulo. The latest inference rules aim to propagate the outermost modulo computation in expressions of the form e **mod** $[l, u[$ into the sub-expression e. The premises of these conditions are not mutually exclusive. They are displayed according to their levels of priority. That is, only the first inference rule that can be applied is applied. In all following rules, any alarms encountered during the translation of sub-expressions are propagated, but no extra alarms are forwarded.

We begin with the case of a sum of two expressions such that the modulo specifications of the two abstract translations of the operands are compatible with the outermost modulo computation of the main expression. This compatibility is checked thanks to the notion of k-splittability. If they are compatible, the modulo specifications of the translations of both sub-expressions are discarded. Then, the abstract counterpart of the sum is used. Lastly, the resulting abstract expression is simplified by the means of the **reduce** parametric operator. This is formalized in the following inference rule.

$$\textsc{ModPlusExpr}\ \frac{\begin{array}{c}(e_1^\sharp, m_1, \Omega_1) \stackrel{\text{def}}{=} (\!(e_1)\!)R^\sharp \\ (e_2^\sharp, m_2, \Omega_2) \stackrel{\text{def}}{=} (\!(e_2)\!)R^\sharp \qquad m_1 \in \mathbb{S}(u_3 - l_3) \qquad m_2 \in \mathbb{S}(u_3 - l_3)\end{array}}{(\!((e_1 + e_2) \textbf{ mod } [l_3, u_3[ )\!)R^\sharp \stackrel{\text{def}}{=} (\textbf{reduce}(e_1^\sharp \boxplus e_2^\sharp)R^\sharp, [l_3, u_3[, \Omega_1 \cup \Omega_2)}$$

The case of a product between two sub-expressions works exactly the same way. This is formalized in the following inference rule.

$$\textsc{ModMultExpr}\ \frac{\begin{array}{c}(e_1^\sharp, m_1, \Omega_1) \stackrel{\text{def}}{=} (\!(e_1)\!)R^\sharp \\ (e_2^\sharp, m_2, \Omega_2) \stackrel{\text{def}}{=} (\!(e_2)\!)R^\sharp \qquad m_1 \in \mathbb{S}(u_3 - l_3) \qquad m_2 \in \mathbb{S}(u_3 - l_3)\end{array}}{(\!((e_1 \times e_2) \textbf{ mod } [l_3, u_3[ )\!)R^\sharp \stackrel{\text{def}}{=} (\textbf{reduce}(e_1^\sharp \boxtimes e_2^\sharp)R^\sharp, [l_3, u_3[, \Omega_1 \cup \Omega_2)}$$

For any other expression, or when the modulo computations are not compatible, the sub-expression e of the modulo computation is translated, which provides the specifications of a potential inner modulo computation m. Then, the application of the modulo m is followed by the application of the initial modulo $[l, u[$. We now introduce three cases, according to specific properties of m and $[l, u[$ to simplify these modulo computations.

As seen in Prop.1, when one modulo computation follows another, the inner one can be discarded provided that the outer one is compatible with it. The compatibility between those modulo computations is checked thanks to the notion of k-splittability. This is formalized in the following inference rule.

$$\textsc{ModIdentity}\ \frac{(e^\sharp, m, \Omega_e) \stackrel{\text{def}}{=} (\!| e |\!) R^\sharp \qquad m \in \mathbb{S}(u-l)}{(\!| e\ \textbf{mod}\ [l,u[\ |\!) R^\sharp \stackrel{\text{def}}{=} (e^\sharp, [l,u[, \Omega_e)}$$

As seen in Prop. 3, in some cases, the outermost modulo interval is large enough and compatible with the modular ring of the translated sub-expression. This happens when elements of the second interval are only translated when applying the modulo on the first interval. In such a case, we return the sub-expression without forgetting to translate the bounds of its modular ring. We can then discard the outermost modulo of the main expression. This is formalized in the following inference rule.

$$\textsc{ModTranslation}\ \frac{\begin{array}{c}(e^\sharp, [l',u'[, \Omega_e) \stackrel{\text{def}}{=} (\!| e |\!) R^\sharp \\ \alpha \stackrel{\text{def}}{=} l' \bmod [l,u[\qquad \alpha + u' - l' \leq u \qquad (u'-l') \mid (\alpha - l')\end{array}}{(\!| e\ \textbf{mod}\ [l,u[\ |\!) R^\sharp \stackrel{\text{def}}{=} (e^\sharp, [\alpha, \alpha + u' - l'[, \Omega_e)}$$

When no other rules can be applied the primitive rmMod is used to eliminate the modulo specifications from the translated sub-expression, and the result is returned along the outermost modulo from the main expression as modular specifications. This is formalized in the following inference rule.

$$\textsc{ModIdentityNoMod}\ \frac{(e^\sharp, m, \Omega_e) \stackrel{\text{def}}{=} (\!| e |\!) R^\sharp}{(\!| e\ \textbf{mod}\ [l,u[\ |\!) R^\sharp \stackrel{\text{def}}{=} (\text{rmMod}(e^\sharp, m) R^\sharp, [l,u[, \Omega_e)}$$

The following theorem states that the evaluation of a translated expression keeps all the possible values of the evaluation of the original expression. Moreover, the set of potential error alarms returned by the translation contains at least all the potential error alarms of the evaluation of the original expression. This is a stronger statement than the one needed to rewrite expressions, that corresponds to the corollary below, because the error alarms that may be yield by the evaluation of the abstract expression are discarded.

Theorem 4. *For all abstract element $R^\sharp$ and every expression $e \in expr$, with $\langle R_0, \Omega_0 \rangle \stackrel{\text{def}}{=} \gamma(R^\sharp)$ and $(e^\sharp, m, \Omega_e) \stackrel{\text{def}}{=} (\!| e |\!) R^\sharp$, then*

$$\forall \rho \in R_0,\ [\![e]\!]^\Omega \rho \subseteq \Omega_e\ \wedge\ [\![e]\!]^V \rho \subseteq \begin{cases} [\![\text{toExpr}(e^\sharp)]\!]^V \rho & \textit{if } m = \mathbb{Z}, \\ [\![\text{toExpr}(e^\sharp)\ \textbf{\textit{mod}}\ [l,u[\ ]\!]^V \rho & \textit{if } m = [l,u[. \end{cases}$$

*By abuse of notation, we allow the syntactic sugar e **mod** $\mathbb{Z}$ that represents e.*

Corollary 1 (of theorem 4). *For all abstract element $R^\sharp$ and expression $e \in$ expr, with $\langle R_0, \Omega_0 \rangle \overset{\text{def}}{=} \gamma(R^\sharp)$, and $(e^\sharp, m, \Omega_e) \overset{\text{def}}{=} (\!(e)\!) R^\sharp$, the following rewriting property holds:*

$$\langle R_0, \Omega_0 \rangle \vDash e \preccurlyeq_{\Omega_e} \text{toExpr}(e^\sharp) \; \boldsymbol{mod} \; m.$$

6.3 Integration with a Numerical Abstract Domain

We introduce a new numerical abstract domain with expression abstraction, noted $\mathcal{D}^\sharp_{\mathcal{L}}$, that is identical to $\mathcal{D}^\sharp$ except for the assignment and guard statements.

For any expression $e \in$ *expr* and any abstract element $R^\sharp \in \mathcal{D}^\sharp$, let us denote $(e^\sharp, m, \Omega_e) \overset{\text{def}}{=} (\!(e)\!) R^\sharp$. Then,

$$\textsc{assign}^\sharp{}_{\mathcal{L}}(X, e) R^\sharp \overset{\text{def}}{=} \langle \varnothing, \Omega_e \rangle \sqcup \textsc{assign}^\sharp(X, \text{toExpr}(\text{rmMod}(e^\sharp, m) R^\sharp)) R^\sharp$$
$$\textsc{guard}^\sharp{}_{\mathcal{L}}(e, \bowtie) R^\sharp \overset{\text{def}}{=} \langle \varnothing, \Omega_e \rangle \sqcup \textsc{guard}^\sharp(\text{toExpr}(\text{rmMod}(e^\sharp, m) R^\sharp), \bowtie) R^\sharp$$

The soundness of these rewritings comes from Theorems. 1 and 3, as well as Corollary. 1.

7 Instantiation of the generic framework

In this section, we provide more explicit definitions for the ι and **reduce** primitives of our parametric abstraction. The other components are supposed to be defined in an underlying domain.

7.1 Intervalization

During the expression abstraction, the ι primitive is used multiple times, either to verify that modulo computations and bound checks can be safely suppressed, or to check that simplifications can be performed, as in the DivPosConst inference rule. Thus, it appears that the more this primitive is precise, the more translations of expression will be precise.

A first possibility would be to represent the possible values of every expression by intervals over $\mathbb{Z}$ as presented in [4]. However, this method lacks the ability to represent relations between variables, which can be necessary to simplify modulo computations. For instance, in the program example introduced at the beginning of the paper Fig. 1, in order to suppress the modulo computations it is necessary to check that $X \geq Y$ holds. Thus, a domain able to represent the range of variables and the inequalities between pairs of variable, as the pentagon domain [17], is enough for our current study cases. However, it would be possible to use more precise abstract domains such as the difference bound matrices domain [19] that detects upper-bounds of the difference between pairs of variables, the octagon abstract domain [20] that handles inequalities of the form $\pm X \pm Y \leq c$ with X, Y variables and c a constant, or the polyhedron abstract domain [8] that keeps trace of linear inequality properties. Although using relational domains might be costly, it is possible to limit it by restraining the number of variables involved in the numerical constraints by a method named packing [2].

7.2 Simplification of abstract expressions

We now introduce a **reduce** implementation, denoted as $\mathbf{reduce}_0$, that is a heuristic which attempts to simplify abstract expressions without concealing potential error alarms. The purpose of this function is to achieve maximum expression canonization by using linear forms whenever possible.

The reduction $\mathbf{reduce}_0(e^\sharp)R^\sharp$ of an abstract expression $e^\sharp$, in the context of an abstract element $R^\sharp$, is defined inductively and by cases, by the means of a set of inference rules. Their premises are mutually exclusive, except for the NoReduce rule that is used only when no other rule can be applied.

Addition. When $\boxed{\mathcal{L}}(0)$ (that is the translation of 0_{cst}) is summed with another abstract expression, the latter abstract expression is returned. This is because 0_{cst} is a neutral element of $(expr, +)$. Also, when linear forms are summed, we return the canonical linear form of their sum. Lastly, when a linear form is added to a convex join expression, the addition is distributed over the operands of the convex join. In such case, it is possible to reduce both the new additions and the resulting abstract expression in order to simplify them. Those behaviors are formalized in the three following inference rules. If neither of the three can be applied, no reduction is performed.

$$\textsc{PlusZero}\ \frac{i, j \in \{1,2\} : i \neq j \qquad e_i^\sharp = \boxed{\mathcal{L}}(0)}{\mathbf{reduce}_0(e_1^\sharp \boxplus e_2^\sharp)R^\sharp \overset{\text{def}}{=} e_j^\sharp}$$

$$\textsc{PlusLinearForms}\ \frac{\begin{array}{cc} e_1^\sharp = \boxed{\mathcal{L}}\left(a_0 + \sum_{X_i \in \mathcal{V}} a_i X_i\right) & e_1^\sharp \neq \boxed{\mathcal{L}}(0) \\ e_2^\sharp = \boxed{\mathcal{L}}\left(b_0 + \sum_{X_i \in \mathcal{V}} b_i X_i\right) & e_2^\sharp \neq \boxed{\mathcal{L}}(0) \end{array}}{\mathbf{reduce}_0(e_1^\sharp \boxplus e_2^\sharp)R^\sharp \overset{\text{def}}{=} \boxed{\mathcal{L}}\left((a_0 + b_0) + \sum_{X_i \in \mathcal{V}} (a_i + b_i) X_i\right)}$$

$$\textsc{PlusConvexJoin}\ \frac{\begin{array}{c} i, j \in \{1,2\} : i \neq j \\ e_i^\sharp = e_{i,1}^\sharp \boxed{\Cup}\, e_{i,2}^\sharp \qquad e_j^\sharp = \boxed{\mathcal{L}}\left(\alpha_0 + \sum_{X_i \in \mathcal{V}} a_i X_i\right) \qquad e_j^\sharp \neq \boxed{\mathcal{L}}(0) \\ e_1'^\sharp \overset{\text{def}}{=} \mathbf{reduce}_0(e_{i,1}^\sharp \boxplus e_j^\sharp)R^\sharp \qquad e_2'^\sharp \overset{\text{def}}{=} \mathbf{reduce}_0(e_{i,2}^\sharp \boxplus e_j^\sharp)R^\sharp \end{array}}{\mathbf{reduce}_0(e_1^\sharp \boxplus e_2^\sharp)R^\sharp \overset{\text{def}}{=} \mathbf{reduce}_0(e_1'^\sharp \boxed{\Cup}\, e_2'^\sharp)R^\sharp}$$

Multiplication. Multiplying a linear form by a constant is the same thing as multiplying the linear form component-wise to its coefficients. One could think this reduction could hide potential errors when the multiplication is performed with 0. However, this is not possible because evaluation of linear forms does not trigger errors. Also, when a linear form is multiplied by a convex join expression, the multiplication is distributed over the operands of the convex join. In such case, it is possible to reduce both the new products and the resulting abstract expression in order to simplify them. Those reductions are formalized in the two following inference rules. If neither of the two can be applied, no reduction is performed.

$$\textsc{MultConst}\ \frac{i, j \in \{1,2\} : i \neq j \qquad e_i^\sharp = \boxed{\mathcal{L}}\left(a_0 + \sum_{X_i \in \mathcal{V}} a_i X_i\right) \qquad e_j^\sharp = \boxed{\mathcal{L}}(b_0)}{\mathbf{reduce}_0(e_1^\sharp \boxtimes e_2^\sharp)R^\sharp \stackrel{\text{def}}{=} \boxed{\mathcal{L}}\left((a_0 \times b_0) + \sum_{X_i \in \mathcal{V}} (a_i \times b_0) X_i\right)}$$

$$\textsc{MultConvexJoin}\ \frac{\begin{array}{c} i, j \in \{1,2\} : i \neq j \\ e_i^\sharp = e_{i,1}^\sharp \boxed{\Cup} e_{i,2}^\sharp \qquad e_j^\sharp = \boxed{\mathcal{L}}\left(\alpha_0 + \sum_{X_i \in \mathcal{V}} a_i X_i\right) \\ e_1'^\sharp \stackrel{\text{def}}{=} \mathbf{reduce}_0(e_{i,1}^\sharp \boxtimes e_j^\sharp)R^\sharp \qquad e_2'^\sharp \stackrel{\text{def}}{=} \mathbf{reduce}_0(e_{i,2}^\sharp \boxtimes e_j^\sharp)R^\sharp \end{array}}{\mathbf{reduce}_0(e_1^\sharp \boxtimes e_2^\sharp)R^\sharp \stackrel{\text{def}}{=} \mathbf{reduce}_0(e_1'^\sharp \boxed{\Cup} e_2'^\sharp)R^\sharp}$$

Division. The only rule introduced for specific reduction of abstract division expressions is the division of a linear form by a nonzero constant that divides all the coefficients of the linear form. In such a case, the resulting linear form is the original one divided component-wise by the coefficient. Like for the reduction rules of multiplication expressions, no errors can be hidden during this process because evaluation of linear forms does not trigger errors. If it cannot be applied, no reduction is performed.

$$\textsc{DivConst}\ \frac{e_2^\sharp = \boxed{\mathcal{L}}(b_0) \qquad b \neq 0 \qquad e_1^\sharp = \boxed{\mathcal{L}}\left((a_0 \times b_0) + \sum_{X_i \in \mathcal{V}} (a_i \times b_0) X_i\right)}{\mathbf{reduce}_0(e_1^\sharp \boxslash e_2^\sharp)R^\sharp \stackrel{\text{def}}{=} \boxed{\mathcal{L}}\left(a_0 + \sum_{X_i \in \mathcal{V}} a_i X_i\right)}$$

Convex Join. The convex join operator abstracts an idempotent, associative, and commutative operator. This can be exploited in the three following rules. If none of the three can be applied, no reduction is performed.

$$\textsc{ConvexJoinIdem1}\ \frac{e_1^\sharp = e_2^\sharp}{\mathbf{reduce}_0(e_1^\sharp \boxed{\Cup} e_2^\sharp)R^\sharp \stackrel{\text{def}}{=} e_1^\sharp}$$

$$\textsc{ConvexJoinIdem2}\ \frac{i, j \in \{1,2\} : i \neq j \qquad e_i^\sharp = e_{i,1}^\sharp \boxed{\Cup} e_{i,2}^\sharp \qquad e_j^\sharp = e_{i,1}^\sharp \vee e_j^\sharp = e_{i,2}^\sharp}{\mathbf{reduce}_0(e_1^\sharp \boxed{\Cup} e_2^\sharp)R^\sharp \stackrel{\text{def}}{=} e_i^\sharp}$$

$$\textsc{ConvexJoinIdem3}\ \frac{\begin{array}{c} e_1^\sharp \neq e_2^\sharp \qquad e_1^\sharp = e_{1,1}^\sharp \boxed{\Cup} e_{1,2}^\sharp \qquad e_2^\sharp = e_{2,1}^\sharp \boxed{\Cup} e_{2,2}^\sharp \\ i, j \in \{1,2\} : i \neq j \qquad e_{1,i}^\sharp = e_{2,1}^\sharp \vee e_{1,i}^\sharp = e_{2,2}^\sharp \end{array}}{\mathbf{reduce}_0(e_1^\sharp \boxed{\Cup} e_2^\sharp)R^\sharp \stackrel{\text{def}}{=} \mathbf{reduce}_0(e_{1,j}^\sharp \boxed{\Cup} e_2^\sharp)R^\sharp}$$

```c
unsigned int x, a;
int16_t b;
if (a <= x && x-a <= 256 && b >= 0) {
  int16_t r = ((x - a) * b) >> 8;
  assert(0 <= r && r <= b);
}
```

(a) C language

```
if $X + -A \geq 0$ then
  if $X + -A \leq 2^8$ then
    if $B \geq 0$ then
      $R \leftarrow ((X + -A) \times B) \,/\, 2^8$;
      // $0 \leq R \leq B$
    endif
  endif
endif
```

(b) Article's language (simplified)

Fig. 10: Second example of linear interpolation computation.

Default rule. As described earlier, if no other reduction rule can be applied, the abstract expression is returned unmodified. This is formalized in the following inference rule that has no premises.

$$\textsc{NoReduce}\ \frac{}{\mathbf{reduce}_0(e^\sharp)R^\sharp \stackrel{\text{def}}{=} e^\sharp}$$

The soundness of the reduction rules of $\mathbf{reduce}_0$ is stated in the following theorem.

Theorem 5. *The* $\boldsymbol{reduce}_0$ *operator we introduced is a* ***reduce*** *operator as described in Def. 5.8.*

7.3 Linear Interpolation

One advantage of handling abstract expressions in the **reduce** function is that they are potentially simpler than the original expressions (*e.g.*, without modulo computations). Moreover, this function is applied during several stages of the reduction. Thus, it is possible to introduce new reduction rules that try to match patterns in order to simplify the matched abstract expressions.

We illustrate this method with the *interpolation detection step* that aims at finding and simplifying linear interpolation patterns. This step consists in the introduction of two new inference rules in $\mathbf{reduce}_0$. The first one matches a product of a linear form $\boxed{\mathcal{L}}(X - A)$ and an abstract expression $e_z^\sharp$, quotient by a linear form $\boxed{\mathcal{L}}(B - A)$ with $A, B, X \in \mathcal{V}$. It claims the quotient can be reduced to $\boxed{\mathcal{L}}(0)\ \boxed{\uplus}\ e_z^\sharp$ (or more precisely its reduction) when the denominator is nonzero and X is between A and B. This is the rule that can be used in the program example given at the beginning of the paper in Fig. 3. The second inductive rule introduced is quite similar, but the denominator is a strictly positive constant. Under the condition that $X - A$ is between 0 and kd, with k the maximum nonzero integer that verifies the property (that exists because $\mathbb{N}^*$ is well-founded), the reduction rule returns $\boxed{\mathcal{L}}(0)\ \boxed{\uplus}\ (e_z^\sharp \boxtimes \boxed{\mathcal{L}}(k))$ (modulo two reductions). This rule can, for example, be used to prove that, in Fig. 10, $0 \leq R \leq B$ holds if R is assigned. If these induction rules cannot be applied, the usual default rule $\textsc{NoReduce}$ is used.

$$\textsc{Interp1}\ \frac{\begin{array}{c} y, z \in \{1,2\} : y \neq z \\ X, A, B \in \mathcal{V} \qquad e_y^\sharp \stackrel{\text{def}}{=} \boxed{\mathcal{L}}(X - A) \qquad \iota(B - A)R^\sharp \subseteq [1, +\infty[\\ \iota(X - A)R^\sharp \subseteq [0, +\infty[\qquad \iota(B - X)R^\sharp \subseteq [0, +\infty[\end{array}}{\mathbf{reduce}_0((e_1^\sharp \boxtimes e_2^\sharp) \boxslash \boxed{\mathcal{L}}(B - A))R^\sharp \stackrel{\text{def}}{=} \mathbf{reduce}_0(\boxed{\mathcal{L}}(0) \boxed{\Cup}\, e_z^\sharp)R^\sharp}$$

$$\textsc{Interp2}\ \frac{\begin{array}{c} y, z \in \{1,2\} : y \neq z \qquad d \in \mathbb{N}^* \qquad X, A \in \mathcal{V} \\ e_y^\sharp = \boxed{\mathcal{L}}(X - A) \qquad k \stackrel{\text{def}}{=} \min\{\, k \in \mathbb{N}^* \mid \iota(X - A)R^\sharp \subseteq [0, kd] \,\} \\ e_z'^\sharp \stackrel{\text{def}}{=} \mathbf{reduce}_0(e_z^\sharp \boxtimes \boxed{\mathcal{L}}(k))R^\sharp \end{array}}{\mathbf{reduce}_0((e_1^\sharp \boxtimes e_2^\sharp) \boxslash \boxed{\mathcal{L}}(d))R^\sharp \stackrel{\text{def}}{=} \mathbf{reduce}_0(\boxed{\mathcal{L}}(0) \boxed{\Cup}\, e_z'^\sharp)R^\sharp}$$

In this section, we have introduced a reduction heuristic over abstract expressions. We have then presented that we can take advantage of the simplified form of the abstract expressions to recognize patterns, such as linear interpolations. Even if they allow some flexibility (*e.g.*, commutation of the operands in multiplications), the capability to recognize all linear interpolations can cause the number of rules to explode. It can then be interesting to memoize the result of pattern recognition so that further iterations would explore fewer cases.

8 Implementation presentation

All the methods we have described have been implemented in the Astrée [1,2] static analyzer but also in an extra toy abstract interpreter of C code [3] we submitted along this article to emphasize our work. Some implementation details and results of our artifact are detailed in this section.

The approach we have presented in this paper is sensitive to program transformations, and particularly to the usage of temporary variables, as in Fig. 2 In the implementations we also use adopted the *Symbolic Constant Propagation* methods [21] to eliminate them by propagating and simplifying the expressions assigned to them. A strategy has to be provided to the symbolic constant propagation domain to decide which variable substitute by its expression. The one we currently use consists in propagating the expressions as soon as they still contain variables.

The artifact has been evaluated on several code excerpts from industrial code. The improvement of both the analysis time and the number of false-alarms returned by the analyzer when adding the rewriting abstract domain that we presented is summarized in Fig. 11. The comparison has been made using three different underlying abstract domains implemented in the library Apron [15]: intervals (boxes), octagons, and polyhedra. If the analysis times out (after 5 seconds), the corresponding bars are hatched and the bar height is set to the height of the graph, so other bars are not flattened.

The C code excerpts that have been tested are the following ones:

– `fig1.c`, `fig2.c`, `fig3.c` and `fig10.c` respectively correspond to Figs. 1, 2, 3, and 10,
– `fig1_promo.c` and `fig2_promo.c` respectively correspond to Figs. 1 and 2 with the extra usage of integer promotion instead of `unsigned` data type,

- `fig3+mod.c` corresponds to Fig. 3 with the extra usage of variable differences as introduced in Fig. 1,
- `promo{1,2,3}.c` present different counter-intuitive effects of integer promotion in C code,
- `div0.c` is a demonstration that rewritings do not hide potential errors, in particular divisions by zero that are discarded during the rewriting,
- `bilinear.c` computes a bilinear interpolation that consists in 8 nested linear interpolations. Its exact range then has to be proven.

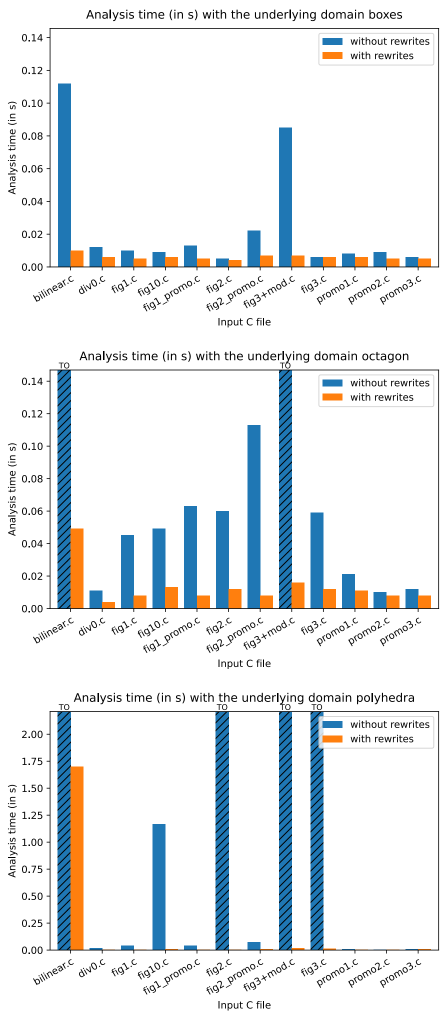
The results of our artifact 11 show that, as soon as the underlying abstract domain is able to prove obligation inequalities, the symbolic domain is able to eliminate inner modulo computations and discover more precise numerical properties. In general, the overhead cost is compensated by the fact that, due to the increase of accuracy, the time spent in the other domains is reduced.

9 Conclusion

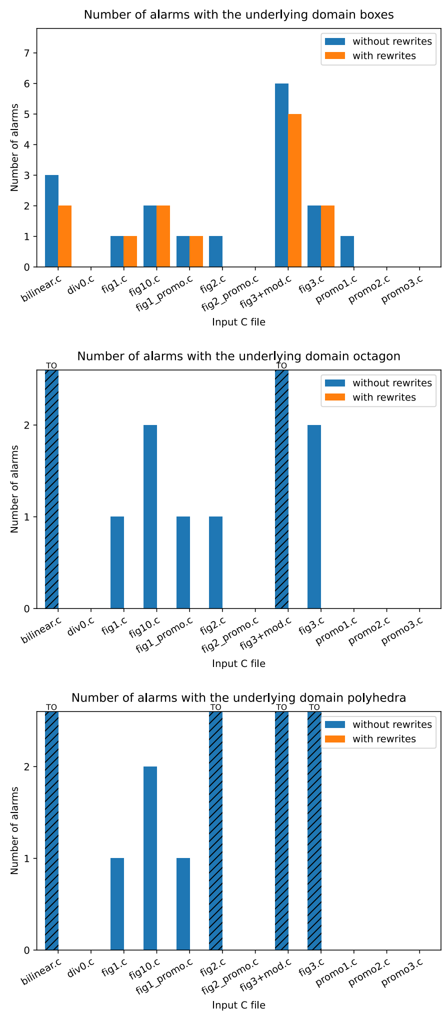
We have proposed in this article a method to safely rewrite arithmetic expressions into simpler ones. In particular, modulo computations, that are frequent in the semantics of real-world programming languages, are safely discarded ; either by proving that they can be precisely described, or by replacing them conservatively. Then, a reduction operator has been introduced to achieve maximum expression canonization by using linear forms to the fullest extent possible. Those reductions allow us to match and simplify expression patterns, such as linear interpolations, for a low cost. Nevertheless, this technique remains general and could be used with other patterns. This method has been implemented within the ASTRÉE static analyzer, and a toy abstract interpreter that is available. An evaluation of the accuracy and the overhead cost induced by the new abstract domain has been presented and supports its effectiveness in the presence of modulo computations. The cost is generally compensated by the fact that, due to the increase of accuracy, the time spent in the other domains is reduced.

The reduction operator can be easily tuned to adapt to a wider class of interpolation scheme or other application domains. We would like to investigate further the accuracy of our analysis in a wider context and develop further refinement of the reduction operator.

Acknowledgements We thank the anonymous referees for their constructive comments and suggestions. Furthermore, we extend our gratitude to Josselin Giet, Marc Chevalier and Antoine Miné for granting us permission to publish a toy abstract interpreter they created, which served as the foundation for our artifact.



(a) Analysis time comparison.



(b) Number of false alarms comparison.

Fig. 11: Comparison of the analysis of C files with and without the usage of the rewriting abstract domain, respectively with the interval, octagon, and polyhedron abstract domain as the underlying one.

References

1. Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: Design and implementation of a special-purpose static program analyzer for safety-critical real-time embedded software. In: The Essence of Computation, Complexity, Analysis, Transformation. Springer (2002). https://doi.org/10.1007/3-540-36377-7_5
2. Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: A static analyzer for large safety-critical software. In: Programming Language Design and Implementation. ACM (2003). https://doi.org/10.1145/781131.781153
3. Boillot, J., Feret, J.: Artifact for "symbolic transformation of expressions in modular arithmetic" (2023). https://doi.org/10.5281/zenodo.8186873
4. Cousot, P., Cousot, R.: Static determination of dynamic properties of programs. In: International Symposium on Programming. Dunod (1976). https://doi.org/10.1145/390019.808314
5. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Principles of Programming Languages. ACM (1977). https://doi.org/10.1145/512950.512973
6. Cousot, P., Cousot, R.: Abstract interpretation and application to logic programs. Journal of Logic Programming (1992). https://doi.org/10.1016/0743-1066(92)90030-7
7. Cousot, P., Cousot, R.: Comparing the Galois connection and widening/narrowing approaches to abstract interpretation, invited paper. In: Programming Language Implementation and Logic Programming. Springer (1992). https://doi.org/10.1007/3-540-55844-6_142
8. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: Principles of Programming Languages. ACM (1978). https://doi.org/10.1145/512760.512770
9. Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., Zadeck, F.K.: Efficiently computing static single assignment form and the control dependence graph. Transactions on Programming Languages and Systems (1991). https://doi.org/10.1145/115372.115320
10. Gallois-Wong, D.: Formalisation en Coq des algorithmes de filtre numérique calculés en précision finie. (Coq formalization of digital filter algorithms computed using finite precision arithmetic). Ph.D. thesis, University of Paris-Saclay, France (2021), https://tel.archives-ouvertes.fr/tel-03202580
11. Granger, P.: Static analysis of arithmetical congruences. International Journal of Computer Mathematics (1989). https://doi.org/10.1080/00207168908803778
12. Granger, P.: Static analysis of linear congruence equalities among variables of a program. In: International Joint Conference on Theory and Practice of Software Development on Colloquium on Trees in Algebra and Programming. Springer (1991). https://doi.org/10.1007/3-540-53982-4_10
13. Halbwachs, N., Péron, M.: Discovering properties about arrays in simple programs. In: Programming Language Design and Implementation. ACM (2008). https://doi.org/10.1145/1375581.1375623
14. ISO: International Standard ISO/IEC 9899:1999. International Organization for Standardization (2007), http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1256.pdf

15. Jeannet, B., Miné, A.: Apron: A library of numerical abstract domains for static analysis. In: Computer Aided Verification. Springer (2009). https://doi.org/10.1007/978-3-642-02658-4_52
16. Journault, M., Miné, A., Monat, R., Ouadjaout, A.: Combinations of reusable abstract domains for a multilingual static analyzer. In: Verified Software. Theories, Tools, and Experiments. Springer (2020). https://doi.org/10.1007/978-3-030-41600-3_1
17. Logozzo, F., Fähndrich, M.: Pentagons: A weakly relational abstract domain for the efficient validation of array accesses. In: Symposium on Applied Computing. ACM (2008). https://doi.org/10.1016/j.scico.2009.04.004
18. Masdupuy, F.: Array abstractions using semantic analysis of trapezoid congruences. In: International Conference on Supercomputing. ACM (1992). https://doi.org/10.1145/143369.143414
19. Miné, A.: A new numerical abstract domain based on difference-bound matrices. In: Programs as Data Objects. Springer (2001). https://doi.org/10.1007/3-540-44978-7_10
20. Mine, A.: The octagon abstract domain. In: Proceedings Eighth Working Conference on Reverse Engineering. IEEE Computer Society (2001). https://doi.org/10.1109/WCRE.2001.957836
21. Miné, A.: Symbolic methods to enhance the precision of numerical abstract domains. In: Verification, Model Checking, and Abstract Interpretation. Springer (2006). https://doi.org/10.1007/11609773_23
22. Müller-Olm, M., Seidl, H.: Analysis of modular arithmetic. In: Programming Languages and Systems. Springer (2005). https://doi.org/10.1007/978-3-540-31987-0_5
23. Simon, A., King, A.: Taming the wrapping of integer arithmetic. In: Static Analysis Symposium. Springer (2007). https://doi.org/10.1007/978-3-540-74061-2_8
24. Venet, A.: The gauge domain: Scalable analysis of linear inequality invariants. In: Computer Aided Verification. Springer (2012). https://doi.org/10.1007/978-3-642-31424-7_15