# Symbolic Methods to Enhance the Precision of Numerical Abstract Domains

Antoine Miné

# Symbolic Methods to Enhance the Precision of Numerical Abstract Domains*

Antoine Miné

École Normale Supérieure, Paris, France,
mine@di.ens.fr,
http://www.di.ens.fr/∼mine

**Abstract** We present lightweight and generic symbolic methods to improve the precision of numerical static analyses based on Abstract Interpretation. The main idea is to simplify numerical expressions before they are fed to abstract transfer functions. An important novelty is that these simplifications are performed on-the-fly, using information gathered dynamically by the analyzer.
A first method, called "linearization," allows abstracting arbitrary expressions into affine forms with interval coefficients while simplifying them. A second method, called "symbolic constant propagation," enhances the simplification feature of the linearization by propagating assigned expressions in a symbolic way. Combined together, these methods increase the relationality level of numerical abstract domains and make them more robust against program transformations. We show how they can be integrated within the classical interval, octagon and polyhedron domains. These methods have been incorporated within the Astrée static analyzer that checks for the absence of run-time errors in embedded critical avionics software. We present an experimental proof of their usefulness.

## 1 Introduction

Ensuring the correctness of software is a difficult but important task, especially in embedded critical applications such as planes or rockets. There is currently a great need for static analyzers able to provide invariants automatically and directly on the source code. As the strongest invariants are not computable in general, such tools need to perform sound approximations at the expense of completeness. In this article, we will only consider the properties of numerical variables and work in the Abstract Interpretation framework. A static analyzer is thus parameterized by a *numerical abstract domain*, that is, a set of computer-representable numerical properties together with algorithms to compute the semantics of program instructions.

There already exist quit a few numerical abstract domains. Well-known examples include the interval domain [5] that discovers variable bounds, and the polyhedron domain [8] for affine inequalities. Each domain achieves some cost

* This work was partially supported by the Astrée RNTL project and the APRON project from the ACI "Sécurité & Informatique."

$X \leftarrow [-10, 20];$
$Y \leftarrow X;$
if $(Y \leq 0)$ { $Y \leftarrow -X;$ }
*// here,* $Y \in [0, 20]$

**Figure1.** Absolute value computation example.

$X \leftarrow [0, 1];$
$Y \leftarrow [0, 0.1];$
$Z \leftarrow [0, 0.2];$
$T \leftarrow (X \times Y) - (X \times Z) + Z;$
*// here,* $T \in [0, 0.2]$

**Figure2.** Linear interpolation computation example.

versus precision balance. In particular, non-relational domains—*e.g.*, the interval domain—are much faster but also much less precise than relational domains—able to discover variable relationships. Although the interval information seem sufficient—it allows expressing most correctness requirements, such as the absence of arithmetic overflows or out-of-bound array accesses—relational invariants are often necessary during the course of the analysis to find tight bounds. Consider, for instance, the program of Fig. 1 that computes the absolute value of $X$. We expect the analyzer to infer that, at the end of the program, $Y \in [0, 20]$. The interval domain will find the coarser result $Y \in [-20, 20]$ because it cannot exploit the information $Y = X$ during the test $Y \leq 0$. The polyhedron domain is precise enough to infer the tightest bounds, but results in a loss of efficiency. In our second example, Fig. 2, $T$ is linearly interpolated between $Y$ and $Z$, thus, we have $T \in [0, 0.2]$. Using plain interval arithmetics, one finds the coarser result $T \in [-0.2, 0.3]$. As the assignment in $T$ is not affine, the polyhedron domain cannot perform any better.

In this paper, we present symbolic enhancement techniques that can be applied to abstract domains to solve these problems and increase their robustness against program transformations. In Fig. 1, our *symbolic constant propagation* is able to propagate the information $Y = X$ and discover tight bounds using only the interval domain. In Fig. 2, our *linearization* technique allows us to prove that $T \in [0, 0.3]$ using the interval domain (this result is not optimal, but still much better than $T \in [-0.2, 0.3]$). The techniques are generic and can be applied to other domains, such as the polyhedron domain. However, the improvement varies greatly from one example to another and enhanced domains do not enjoy best abstraction functions. Thus, our techniques depend upon *strategies*, some of which are proposed in the article.

*Related Work.* Our linearization can be related to *affine arithmetics*, a technique introduced by Vinícius et al. in [16] to refine interval arithmetics by taking into account existing correlations between computed quantities. Both use a symbolic form with linear properties to allow basic algebraic simplifications. The main difference is that we relate directly program variables while affine arithmetics

$$
\begin{array}{lll}
\mathit{expr} ::= X & & X \in \mathcal{V} \\
\quad | \quad [a, b] & & a \in \mathbb{I} \cup \{-\infty\},\ b \in \mathbb{I} \cup \{+\infty\},\ a \leq b \\
\quad | \quad \mathit{expr} \diamond \mathit{expr} & & \diamond \in \{+, -, \times, /\} \\
\\
\mathit{inst} ::= X \leftarrow \mathit{expr} & & X \in \mathcal{V} \\
\quad | \quad \mathit{expr} \bowtie 0\ ? & & \bowtie \in \{=, \neq, <, \leq, \geq, >\}
\end{array}
$$

**Figure3.** Syntax of our simple language.

introduces synthetic variables. This allows us to treat control flow joins and loops, and to interact with relational domains, which is not possible with affine arithmetics. Our linearization was first introduced in [13] to abstract floating-point arithmetics. It is presented here with some improvements—including the introduction of several strategies.

Our symbolic constant propagation technique is similar to the classical constraint propagation proposed by Kildall in [11] to perform optimization. However, scalar constants are replaced with expression trees, and our goal is not to improve the efficiency but the precision of the abstract execution. It is also related to the work of Colby: he introduces, in [4], a language of transfer relations to propagate, combine and simplify, in a fully symbolic way, sequences of transfer functions. We are more modest as we do not handle disjunctions symbolically and do not try to infer symbolic loop invariants. Instead, we rely on the underlying numerical abstract domain to perform most of the semantical job. A major difference is that, while Colby's framework statically transforms the abstract equation system to be solved by the analyzer, our framework performs this transformation on-the-fly and benefits from the information dynamically inferred by the analyzer.

*Overview of the Paper.* The paper is organised as follows. In Sect. 2, we introduce a language—much simplified for the sake of illustration—and recall how to perform a numerical static analysis parameterized by an abstract domain. Sect. 3 then explains how symbolic expression manipulations can be soundly incorporated within the analysis. Two symbolic methods are then introduced: expression linearization, in Sect. 4, and symbolic constant propagation, in Sect. 5. Sect. 6 discusses our practical implementation within the Astrée static analyzer and presents some experimental results. We conclude in Sect. 7.

## 2 Framework

In this section, we briefly recall the classical design of a static analyzer using the Abstract Interpretation framework by Cousot and Cousot [6, 7]. This design is specialised towards the automatic computation of *numerical* invariants, and thus, is parameterized by a numerical abstract domain.

### 2.1 Syntax of the Language

For the sake of presentation, we will only consider in this article a very simplified programming language focusing on manipulating numerical variables.

$$
\begin{array}{lcll}
[\![ X ]\!](\rho) & \stackrel{\text{def}}{=} & \{\ \rho(X)\ \} & \\
[\![ [a,b] ]\!](\rho) & \stackrel{\text{def}}{=} & \{\ x \in \mathbb{I} \mid a \leq x \leq b\ \} & \\
[\![ e_1 \diamond e_2 ]\!](\rho) & \stackrel{\text{def}}{=} & \{\ x \diamond y \mid x \in [\![ e_1 ]\!](\rho),\ y \in [\![ e_2 ]\!](\rho)\ \} \qquad \diamond \in \{+,-,\times\} & \\
[\![ e_1/e_2 ]\!](\rho) & \stackrel{\text{def}}{=} & \{\ \mathit{truncate}(x/y) \mid x \in [\![ e_1 ]\!](\rho),\ y \in [\![ e_2 ]\!](\rho),\ y \neq 0\ \} & \text{if } \mathbb{I} = \mathbb{Z} \\
[\![ e_1/e_2 ]\!](\rho) & \stackrel{\text{def}}{=} & \{\ x/y \mid x \in [\![ e_1 ]\!](\rho),\ y \in [\![ e_2 ]\!](\rho),\ y \neq 0\ \} & \text{if } \mathbb{I} \neq \mathbb{Z}
\end{array}
$$

$$
\begin{array}{lcl}
\{\!| X \leftarrow e |\!\}(R) & \stackrel{\text{def}}{=} & \{\ \rho[X \mapsto v] \mid \rho \in R,\ v \in [\![ e ]\!](\rho)\ \} \\
\{\!| e \bowtie 0\ ?\ |\!\}(R) & \stackrel{\text{def}}{=} & \{\ \rho \mid \rho \in R \text{ and } \exists\, v \in [\![ e ]\!](\rho),\ v \bowtie 0 \text{ holds}\ \}
\end{array}
$$

**Figure4.** Concrete semantics.

We suppose that a program manipulates only a fixed, finite set of $n$ variables, $\mathcal{V} \stackrel{\text{def}}{=} \{V_1, \ldots, V_n\}$, with values within a perfect mathematical set, $\mathbb{I} \in \{\mathbb{Z}, \mathbb{Q}, \mathbb{R}\}$. A program $P \in \mathcal{P}(\mathcal{L} \times \mathit{inst} \times \mathcal{L})$ is a single control-flow graph where nodes are program points, in $\mathcal{L}$, and arcs are labelled by instructions in *inst*. We denote by $e$ the entry program point. As described in Fig. 3, only two types of instructions are allowed: assignments ($X \leftarrow \mathit{expr}$) and tests ($\mathit{expr} \bowtie 0\ ?$), where *expr* are numerical expressions and $\bowtie$ is a comparison operator. In the syntax of expressions, classical numerical constants have been replaced with *intervals* $[a, b]$ with constant bounds—possibly $+\infty$ or $-\infty$. Such intervals correspond to a non-deterministic choice of a new value within the bounds each time the expression is evaluated. This will be key in defining the concept of *expression abstraction* in Sects. 3–5. Moreover, interval constants appear naturally in programs that fetch input values from an external environment, or when modeling rounding errors in floating-point computations.

Affine forms play an important role in program analysis as they are easy to manipulate and appear frequently as program invariants. We enhance affine forms with the non-determinism of intervals by defining *interval affine forms* as the expressions of the form: $[a_0, b_0] + \sum_k ([a_k, b_k] \times V_k)$.

### 2.2 Concrete Semantics of the Language

The *concrete semantics* of a program is the most precise mathematical expression of its behavior. Let us first define an *environment* as a function, in $\mathcal{V} \to \mathbb{I}$, associating a value to each variable. We choose a simple *invariant semantics* that associates to each program point $l \in \mathcal{L}$ the set of all environments $\mathcal{X}_l \in \mathcal{P}(\mathcal{V} \to \mathbb{I})$ that can hold when $l$ is reached. Given an environment $\rho \in (\mathcal{V} \to \mathbb{I})$, the semantics $[\![ \mathit{expr} ]\!](\rho)$ of an expression *expr*, shown in Fig. 4, is the set of values the expression can evaluate to. It outputs a set to account for non-determinism. When $\mathbb{I} = \mathbb{Z}$, the *truncate* function rounds the possibly non-integer result of the division towards an integer by *truncation*, as it is common in most computer languages. Divisions by zero are undefined, that is, return no result; for the sake of simplicity, we have not introduced any error state. The semantics of assignments and tests is defined by *transfer functions* $\{\!| \mathit{inst} |\!\} : \mathcal{P}(\mathcal{V} \to \mathbb{I}) \to \mathcal{P}(\mathcal{V} \to \mathbb{I})$ in Fig. 4. The assignment transfer function returns environments where

one variable has changed its value ($\rho[V \mapsto x]$ denotes the function equal to $\rho$ on $\mathcal{V} \setminus \{V\}$ and that maps $V$ to $x$). The test transfer function filters environments to keep only those that *may* satisfy the test. We can now define the semantics $(\mathcal{X}_l)_{l \in \mathcal{L}}$ of a program $P$ as the smallest solution of the following equation system:

$$\begin{cases} \mathcal{X}_e = & V \to \mathbb{I} \\ \mathcal{X}_l = & \bigcup\limits_{(l',i,l) \in P} \{\!|\, i \,|\!\}(\mathcal{X}_{l'}) \quad \text{when } l \neq e \end{cases} \tag{1}$$

It describes the *strongest invariant* at each program point.

### 2.3 Abstract Interpretation and Numerical Abstract Domains

The concrete semantics is very precise but cannot be computed fully automatically by a computer. We will only try to compute a sound overapproximation, that is, a *superset* of the environments reached by the program. We use Abstract Interpretation [6, 7] to design such an approximation.

*Numerical Abstract Domains.* An analysis is parameterized by a numerical abstract domain that allows representing and manipulating selected subsets of environments. Formally it is defined as:

- a set of computer-representable *abstract* elements $\mathcal{D}^\sharp$,
- a *partial order* $\sqsubseteq^\sharp$ on $\mathcal{D}^\sharp$ to model the relative precision of abstract elements,
- a monotonic *concretization* $\gamma : \mathcal{D}^\sharp \to \mathcal{P}(\mathcal{V} \to \mathbb{I})$, that assigns a concrete property to each abstract element,
- a greatest element $\top^\sharp$ for $\sqsubseteq^\sharp$ such that $\gamma(\top^\sharp) = (\mathcal{V} \to \mathbb{I})$,
- *sound* and computable abstract versions $\{\!|\, inst \,|\!\}^\sharp$ of all transfer functions,
- *sound* and computable abstractions $\cup^\sharp$ and $\cap^\sharp$ of $\cup$ and $\cap$,
- a widening operator $\triangledown^\sharp$ if $\mathcal{D}^\sharp$ has infinite increasing chains.

The *soundness condition* for the abstraction $F^\sharp : (\mathcal{D}^\sharp)^n \to \mathcal{D}^\sharp$ of a $n-$ary operator $F$ is: $F(\gamma(X_1^\sharp), \ldots, \gamma(X_n^\sharp)) \subseteq \gamma(F^\sharp(X_1^\sharp, \ldots, X_n^\sharp))$. It ensures that $F^\sharp$ does not forget any of $F$'s behaviors. It can, however, introduce spurious ones.

*Abstract Analysis.* Given an abstract domain, an abstract version $(1^\sharp)$ of the equation system (1) can be derived as:

$$\begin{cases} \mathcal{X}_e^\sharp = & \top^\sharp \\ \mathcal{X}_l^\sharp \sqsubseteq^\sharp & \bigcup\limits^\sharp_{(l',i,l) \in P} \{\!|\, i \,|\!\}^\sharp(\mathcal{X}_{l'}^\sharp) \quad \text{when } l \neq e \end{cases} \tag{$1^\sharp$}$$

The soundness condition ensures that any solution of $(1^\sharp)$ satisfies $\forall l \in \mathcal{L},\ \gamma(\mathcal{X}_l^\sharp) \supseteq \mathcal{X}_l$. The system can be solved by iterations, using a widening operator $\triangledown^\sharp$ to ensure termination. We refer the reader to Bourdoncle [2] for an in-depth description of possible iteration strategies. The computed $\mathcal{X}_l^\sharp$ is almost never the best abstraction—if it exists—of the concrete solution $\mathcal{X}_l$. Unavoidable losses of precision come from the use of convergence acceleration $\triangledown^\sharp$, non-necessarily best abstract transfer functions, and the fact that the composition of best abstractions is generally not a best abstraction. This last issue explains why even the simplest semantics-preserving program transformations can drastically

affect the quality of a static analysis.

*Existing Numerical Domains.* There exists many numerical abstract domains. We will be mostly interested in those able to express variable bounds. Such abstract domains include the well-known interval domain [5] (able to express invariants of the form $\bigwedge_i\ V_i \in [a_i, b_i]$), and the polyhedron domain [8] (able to express affine inequalities $\bigwedge_i\ \sum_j \alpha_{ij} V_i \geq \beta_j$). More recent domains, in-between these two in terms of cost and precision, include the octagon domain [12] ($\bigwedge_{ij}\ \pm V_i \pm V_j \leq c_{ij}$), the octahedron domain [3] ($\bigwedge_j\ \sum_i \alpha_{ij} V_i \geq \beta_j$ where $\alpha_{ij} \in \{-1, 0, 1\}$), and the Two Variable Per Inequality domain [15] ($\bigwedge_i\ \alpha_i V_{k_i} + \beta_i V_{l_i} \leq c_i$).

## 3 Incorporating Symbolic Methods

We suppose that we are given a numerical abstract domain $\mathcal{D}^\sharp$. The gist of our method is to replace, in the abstract transfer functions $\{\!|\, X \leftarrow e \,|\!\}^\sharp$ and $\{\!|\, e \bowtie 0\ ?\ |\!\}^\sharp$, each expression $e$ with another one $e'$, in a sound way.

*Partial Order on Expressions.* To define formally the notion of sound expression abstraction, we first introduce an approximation order $\preceq$ on expressions. A natural choice is to consider the point-wise ordering of the concrete semantics $[\![\cdot]\!]$ defined in Fig. 4, that is: $e_1 \preceq e_2 \iff^{\text{def}} \forall \rho \in (\mathcal{V} \to \mathbb{I}),\ [\![ e_1 ]\!](\rho) \subseteq [\![ e_2 ]\!](\rho)$. However, requiring the inclusion to hold for *all* environments is quite restrictive. More aggressive expression transformations can be enabled by only requiring soundness with respect to selected sets of environments. Our partial order $\preceq$ is now defined "up to" a set of environments $R \in \mathcal{P}(\mathcal{V} \to \mathbb{I})$:

**Definition 1** $R \models e_1 \preceq e_2 \iff^{\text{def}} \forall \rho \in R,\ [\![ e_1 ]\!](\rho) \subseteq [\![ e_2 ]\!](\rho)$.

We denote by $R \models e_1 = e_2$ the associated equality relation.

*Sound Symbolic Transformations.* We wish now to abstract some transfer function, *e.g.*, $\{\!|\, V \leftarrow e \,|\!\}$, on an abstract environment $R^\sharp \in \mathcal{D}^\sharp$. The following theorem states that, if $e'$ overapproximates $e$ on $\gamma(R^\sharp)$, it is sound to replace $e$ with $e'$ in the abstract transfer functions:

**Theorem 1** *If* $\gamma(R^\sharp) \models e \preceq e'$*, then:*

- $(\{\!|\, V \leftarrow e \,|\!\} \circ \gamma)(R^\sharp) \subseteq (\gamma \circ \{\!|\, V \leftarrow e' \,|\!\}^\sharp)(R^\sharp)$,
- $(\{\!|\, e \bowtie 0\ ?\ |\!\} \circ \gamma)(R^\sharp) \subseteq (\gamma \circ \{\!|\, e' \bowtie 0\ ?\ |\!\}^\sharp)(R^\sharp)$.

## 4 Expression Linearization

Our first symbolic transformation is an abstraction of arbitrary expressions into interval affine forms $i_0 + \sum_k (i_k \times V_k)$, where the $i$'s stand for intervals.

### 4.1 Definitions

*Interval Affine Form Operators.* We first introduce a few operators to manipulate interval affine forms in a symbolic way. Using the classical interval arithmetic

operators—denoted with a $\mathcal{I}$ superscript—we can define point-wisely the addition $\boxplus$ and subtraction $\boxminus$ of affine forms, as well as the multiplication $\boxtimes$ and division $\boxslash$ of an affine form by a constant interval:

**Definition 2**

- $(i_0 + \sum_k i_k \times V_k) \boxplus (i'_0 + \sum_k i'_k \times V_k) \stackrel{\text{def}}{=} (i_0 +^{\mathcal{I}} i'_0) + \sum_k (i_k +^{\mathcal{I}} i'_k) \times V_k$,
- $(i_0 + \sum_k i_k \times V_k) \boxminus (i'_0 + \sum_k i'_k \times V_k) \stackrel{\text{def}}{=} (i_0 -^{\mathcal{I}} i'_0) + \sum_k (i_k -^{\mathcal{I}} i'_k) \times V_k$,
- $i \boxtimes (i_0 + \sum_k i_k \times V_k) \stackrel{\text{def}}{=} (i \times^{\mathcal{I}} i_0) + \sum_k (i \times^{\mathcal{I}} i_k) \times V_k$,
- $(i_0 + \sum_k i_k \times V_k) \boxslash i \stackrel{\text{def}}{=} (i_0 /^{\mathcal{I}} i) + \sum_k (i_k /^{\mathcal{I}} i) \times V_k$.

*where the interval arithmetic operators are defined classically as:*

- $[a,b] +^{\mathcal{I}} [a',b'] \stackrel{\text{def}}{=} [a+a', b+b']$,
- $[a,b] -^{\mathcal{I}} [a',b'] \stackrel{\text{def}}{=} [a-b', b-a']$,
- $[a,b] \times^{\mathcal{I}} [a',b'] \stackrel{\text{def}}{=} [\min(aa', ab', ba', bb'), \max(aa', ab', ba', bb')]$,
- $[a,b] /^{\mathcal{I}} [a',b'] \stackrel{\text{def}}{=}$

$$\begin{cases} [-\infty, +\infty] & \text{if } 0 \in [a',b'] \\ [\min(a/a', a/b', b/a', b/b'),\ \max(a/a', a/b', b/a', b/b')] & \text{when } \mathbb{I} \neq \mathbb{Z} \\ [\lfloor \min(a/a', a/b', b/a', b/b') \rfloor,\ \lceil \max(a/a', a/b', b/a', b/b') \rceil] & \text{when } \mathbb{I} = \mathbb{Z} \end{cases}$$

The following theorem states that these operators are always sound and, in some cases, complete—*i.e.*, $\preceq$ can be replaced by $=$:

**Theorem 2** *For all interval affine forms $l_1$, $l_2$ and interval $i$, we have:*

- $\mathbb{I}^{\mathcal{V}} \models l_1 + l_2 = l_1 \boxplus l_2$,
- $\mathbb{I}^{\mathcal{V}} \models l_1 - l_2 = l_1 \boxminus l_2$,
- $\mathbb{I}^{\mathcal{V}} \models i \times l_1 = i \boxtimes l_1$, *if* $\mathbb{I} \neq \mathbb{Z}$,
- $\mathbb{I}^{\mathcal{V}} \models i \times l_1 \preceq i \boxtimes l_1$, *otherwise,*
- $\mathbb{I}^{\mathcal{V}} \models l_1 / i = l_1 \boxslash i$, *if* $\mathbb{I} \neq \mathbb{Z}$ *and* $0 \notin i$,
- $\mathbb{I}^{\mathcal{V}} \models l_1 / i \preceq l_1 \boxslash i$, *otherwise.*

When $\mathbb{I} = \mathbb{Z}$, we must conservatively round upper and lower bounds respectively towards $+\infty$ and $-\infty$ to ensure that Thm. 2 holds. The non-exactness of the multiplication and division can then lead to some precision degradation. For instance, $(X \boxslash 2) \boxtimes 2$ evaluates to $[0,2] \times X$ as, when computing $X \boxslash 2$, the non-integral value $1/2$ must be abstracted into the integral interval $[0,1]$. One solution is to perform all computations in $\mathbb{R}$, keeping in mind that, due to truncation, $l/[a,b]$ should be interpreted when $0 \notin [a,b]$ as $(l \boxslash [a,b]) \boxplus [-1+x, 1-x]$, where $x = 1/\min(|a|,|b|)$. We then obtain the more precise result $X + [-1,1]$.

We now introduce a so-called "intervalization" operator, $\iota$, to abstracts interval affine forms into intervals. Given an abstract environment, it evaluates the affine form using interval arithmetics. Suppose that $\mathcal{D}^\sharp$ provides us with projection operators $\pi_k : \mathcal{D}^\sharp \to \mathcal{P}(\mathbb{I})$ able to return an interval overapproximation for each variable $V_k$. We define $\iota$ as:

**Definition 3** $\iota(i_0 + \sum_k (i_k \times V_k))(R^\sharp) \stackrel{\text{def}}{=} i_0 +^{\mathcal{I}} \sum_k^{\mathcal{I}} (i_k \times^{\mathcal{I}} \pi_k(R^\sharp))$,
*where each $\pi_k(R^\sharp)$ is an interval containing* $\{\ \rho(V_k) \mid \rho \in \gamma(R^\sharp)\ \}$.

The following theorem states that $\iota$ is a sound operator with respect to $R^\sharp$:

**Theorem 3** $\gamma(R^\sharp) \models l \preceq \iota(l)(R^\sharp)$.

As $\pi_k$ performs a non-relational abstraction, $\iota$ incurs a loss of precision whenever $\mathcal{D}^\sharp$ is a relational domain. Consider, for instance $R^\sharp$ such that $\gamma(R^\sharp) = \{\ \rho \in (\{V_1, V_2\} \to [0,1]) \mid \rho(V_1) = \rho(V_2)\ \}$. Then, $[\![\, \iota(V_1 - V_2)(R^\sharp) \,]\!]$ is the constant function $[-1,1]$ while $[\![\, V_1 - V_2 \,]\!]$ is 0.

*Linearization.* The linearization $(\!|\, e \,|\!)(R^\sharp)$ of an arbitrary expression $e$ in an abstract environment $R^\sharp$ can now be defined by structural induction as follows:

**Definition 4**

- $(\!|\, V \,|\!)(R^\sharp) \stackrel{\text{def}}{=} [1,1] \times V$,
- $(\!|\, [a,b] \,|\!)(R^\sharp) \stackrel{\text{def}}{=} [a,b]$,
- $(\!|\, e_1 + e_2 \,|\!)(R^\sharp) \stackrel{\text{def}}{=} (\!|\, e_1 \,|\!)(R^\sharp) \boxplus (\!|\, e_2 \,|\!)(R^\sharp)$,
- $(\!|\, e_1 - e_2 \,|\!)(R^\sharp) \stackrel{\text{def}}{=} (\!|\, e_1 \,|\!)(R^\sharp) \boxminus (\!|\, e_2 \,|\!)(R^\sharp)$,
- $(\!|\, e_1 / e_2 \,|\!)(R^\sharp) \stackrel{\text{def}}{=} (\!|\, e_1 \,|\!)(R^\sharp) \boxslash \iota((\!|\, e_2 \,|\!)(R^\sharp))(R^\sharp)$,
- $(\!|\, e_1 \times e_2 \,|\!)(R^\sharp) \stackrel{\text{def}}{=} \begin{cases} \textit{either} & \iota((\!|\, e_1 \,|\!)(R^\sharp))(R^\sharp) \boxtimes (\!|\, e_2 \,|\!)(R^\sharp) \\ \textit{or} & \iota((\!|\, e_2 \,|\!)(R^\sharp))(R^\sharp) \boxtimes (\!|\, e_1 \,|\!)(R^\sharp) \end{cases}$ *(see Sect. 4.3)*

The $\iota$ operator is used to deal with non-linear constructions: the right argument of a division and either argument of a multiplication are intervalized. As a consequence of Thms. 2 and 3, our linearization is sound:

**Theorem 4** $\gamma(R^\sharp) \models e \preceq (\!|\, e \,|\!)(R^\sharp)$.

Obviously, $(\!|\, \cdot \,|\!)$ generally incurs a loss of precision with respect to $\preceq$. Also, $(\!|\, e \,|\!)$ is not monotonic in its $e$ argument. Consider for instance $X/X$ in the environment $R^\sharp$ such that $\pi_X(R^\sharp) = [1, +\infty]$. Although $\gamma(R^\sharp) \models X/X \preceq [1,1]$, we do not have $\gamma(R^\sharp) \models (\!|\, X/X \,|\!)(R^\sharp) \preceq (\!|\, [1,1] \,|\!)(R^\sharp)$ as $(\!|\, X/X \,|\!)(R^\sharp) = [0,1] \times X$. It is important to note that there is no useful notion of *best abstraction* of expressions for $\preceq$.

### 4.2 Integration With a Numerical Abstract Domain

Given an abstract domain, $\mathcal{D}^\sharp$, we can now derive a new abstract domain with linearization, $\mathcal{D}^\sharp_{\mathcal{L}}$, identical to $\mathcal{D}^\sharp$ except for the following transfer functions:

$$\{\!|\, V \leftarrow e \,|\!\}^\sharp_{\mathcal{L}}(R^\sharp) \stackrel{\text{def}}{=} \{\!|\, V \leftarrow (\!|\, e \,|\!)(R^\sharp) \,|\!\}^\sharp(R^\sharp)$$
$$\{\!|\, e \bowtie 0\ ?\, |\!\}^\sharp_{\mathcal{L}}(R^\sharp) \stackrel{\text{def}}{=} \{\!|\, (\!|\, e \,|\!)(R^\sharp) \bowtie 0\ ?\, |\!\}^\sharp(R^\sharp)$$

The soundness of these transfer functions is guaranteed by Thms. 1 and 4.

*Application to the Interval Domain.* As all non-relational domains, the interval domain [5], is not able to exploit the fact that the same variable occurs several times in an expression. Our linearization performs some symbolic simplification, and so, is able to partly correct this problem. Consider, for instance, the assignment $\{\!|\, Y \leftarrow 3 \times X - X \,|\!\}$ in an abstract environment such that $X \in [a,b]$. The regular interval domain $\mathcal{D}^{\mathcal{I}}$ will assign $[3a - b, 3b - a]$ to $Y$, while $\mathcal{D}^{\mathcal{I}}_{\mathcal{L}}$ will assign $[2a, 2b]$ as $(\!|\, 3 \times X - X \,|\!)(R^\sharp) = 2 \times X$. This last answer is strictly more precise whenever $a \neq b$. Using the exactness of Thm. 2, one can prove that, when $\mathbb{I} \neq \mathbb{Z}$, the assignment in $\mathcal{D}^{\mathcal{I}}_{\mathcal{L}}$ is always more precise than in $\mathcal{D}^{\mathcal{I}}$. This may not be the case for a test, or when $\mathbb{I} = \mathbb{Z}$.

*Application to the Octagon Domain.* The octagon domain [12] is more precise

than the interval one, but it is more complex. As a consequence, it is quite difficult to design abstract transfer functions for non-linear expressions. This problem can be solved by using our linearization in combination with the efficient and rather precise interval affine form abstract transfer functions proposed in our previous work [14]. The octagon domain with linearization is able to prove, for instance, that, after the assignment $X \leftarrow T \times Y$ in an environment such that $T \in [-1, 1]$, we have $-Y \leq X \leq Y$.

*Application to the Polyhedron Domain.* The polyhedron domain [8] is more precise than the octagon domain but cannot deal with full interval affine forms—only the constant coefficient may safely be an interval. To solve this problem, we introduce a function $\mu$ to abstract interval affine forms further by making all variable coefficients singletons. For the sake of conciseness, we give a formula valid only for $\mathbb{I} \neq Z$ and finite interval bounds:

**Definition 5**

$$\mu\left([a_0, b_0] + \textstyle\sum_k [a_k, b_k] \times V_k\right)(R^\sharp) \stackrel{\text{def}}{=}$$
$$\left([a_0, b_0] +^{\mathcal{I}} \textstyle\sum_k^{\mathcal{I}} [(a_k - b_k)/2, (b_k - a_k)/2] \times^{\mathcal{I}} \pi_k(R^\sharp)\right) + \textstyle\sum_k ((a_k + b_k)/2) \times V_k$$

$\mu$ works by "distributing" the weight $b_k - a_k$ of each variable coefficient into the constant component, using variable bounds information from $R^\sharp$. One can prove that $\mu$ is sound, that is, $\gamma(R^\sharp) \models l \preceq \mu(l) R^\sharp$.

*Application to Floating-Point Arithmetics.* Real-life programming languages do not manipulate rationals or reals, but floating-point numbers, which are much more difficult to abstract. Pervasive rounding must be taken into account. As most classical properties of arithmetic operators are no longer true, it is generally not safe to feed floating-point expressions to relational domains. One solution is to convert such expressions into real-valued expressions by making rounding explicit. Rounding is highly non-linear but can be abstracted using intervals. For instance, $X + Y$ in the floating-point world can be abstracted into $[1 - \epsilon_1, 1 + \epsilon_1] \times X + [1 - \epsilon_1, 1 + \epsilon_1] \times Y + [-\epsilon_2, \epsilon_2]$ using small constants $\epsilon_1$ and $\epsilon_2$ modeling, respectively, relative and absolute errors. This fits in our linearization framework which can be extended to treat soundly floating-point arithmetics. We refer the reader to related work [13] for more information.

### 4.3 Multiplication Strategies

When encountering a multiplication $e_1 \times e_2$ and neither $(\!|\, e_1 \,|\!)(R^\sharp)$ nor $(\!|\, e_2 \,|\!)(R^\sharp)$ evaluates to an interval, we must intervalize either argument. Both choices are valid, but influence greatly the precision of the result.

*All-Cases Strategy.* A first idea is to try both choices for each multiplication; we get a *set* of linearized expressions. We have no notion of greatest lower bound on expressions, so, we must evaluate a transfer function for all expressions in parallel, and take the intersection $\cap^\sharp$ of the resulting abstract elements in $\mathcal{D}^\sharp$. Unfortunately, the cost is exponential in the number of multiplications in the original expression, hence the need for deterministic strategies that always select

*one* interval affine form.

*Interval-Size Strategy.* A simple strategy is to intervalize the affine form that will yield the narrower interval. This greedy approach tries to limit the amplitude of the non-determinism introduced by multiplications. The extreme case holds when the amplitude of one interval is zero, meaning that the sub-expression is semantically a constant; intervalizing it will not result in any precision loss. Finally, note that the *relative* amplitude $(b-a)/|a+b|$ may be more significant than the absolute amplitude $b-a$ if we want to intervalize preferably expressions that are constant up to some small relative rounding error.

*Simplification-Driven Strategy.* Another idea is to maximize the amount of simplification by not intervalizing, when possible, sub-expressions containing variables appearing in other sub-expressions. For instance, in $X-(Y\times X)$, $Y$ will be intervalized to yield $[1-\max Y, 1-\min Y]\times X$. Unlike the preceding greedy approach, this strategy is global and treats the expression as a whole.

*Homogeneity Strategy.* We now consider the linear interpolation of Fig. 2. In order to achieve the best precision, it is important to intervalize $X$ in both multiplications. This yields $T \leftarrow [0,1]\times Y + [0,1]\times Z$ and we are able to prove that $T \geq 0$—however, we find that $T \leq 0.3$ while in fact $T \leq 0.2$. The interval-size strategy would choose to intervalize $Y$ and $Z$ that have smaller range than $X$, which yields the imprecise assignment $T \leftarrow [-0.2, 0.1]\times X + [0, 0.2]$. Likewise, the simplification-driven strategy may choose to keep $X$ that appears in two sub-expressions and also intervalize both $Y$ and $Z$. To solve this problem, we propose to intervalize the smallest set of variables that makes the expression homogeneous, that is, arguments of $+$ and $-$ operators should have the same degree. In order to make the $(1-X)$ sub-expression homogeneous, $X$ is intervalized. This last strategy is quite robust: it keeps working if we change the assignment into the equivalent $T \leftarrow X\times Y - X\times Z + Z$, or if we consider bi-linear interpolations or interpolations with normalization coefficients.

### 4.4 Concluding Remark

Our linearization is not equivalent to a static program transformation. To cope with non-linearity as best as we can, we exploit the information dynamically inferred by the analysis: first, in the intervalization $\iota$, then, in the multiplication strategy. Both algorithms take as argument the current numerical abstract environment $R^\sharp$. As, dually, the linearization improves the precision of the next computed abstract element, the dynamic nature of our approach ensures a positive feed-back.

## 5 Symbolic Constant Propagation

The automatic symbolic simplification implied by our linearization allows us to gain much precision when dealing with complex expressions, without the burden of designing new abstract domains tailored for them. However, the analysis is

still sensitive to program transformations that decompose expressions and introduce new temporary variables—such as common sub-expression elimination or register spilling. In order to be immune to this problem, one must generally use an expressive, and so, costly, *relational* domain. We propose an alternate, lightweight solution based on a kind of constant domain that tracks assignments dynamically and propagate symbolic expressions within transfer functions.

### 5.1 The Symbolic Constant Domain

*Enriched Expressions.* We denote by $\mathcal{C}$ the set of all syntactic expressions, enriched with one element $\top^{\mathcal{C}}$ denoting 'any value.' The flat ordering $\sqsubseteq^{\mathcal{C}}$ is defined as $X \sqsubseteq^{\mathcal{C}} Y \iff Y = \top^{\mathcal{C}}$ or $X = Y$. The concrete semantics $[\![ \cdot ]\!]$ of Fig. 4 is extended to $\mathcal{C}$ as $[\![ \top^{\mathcal{C}} ]\!](\rho) = \mathbb{I}$. We also use two functions on expression trees: $occ : \mathcal{C} \to \mathcal{P}(\mathcal{V})$ that returns the set of variables occurring in an expressing, and $subst : \mathcal{C} \times \mathcal{V} \times \mathcal{C} \to \mathcal{C}$ that substitutes, in its first argument, every occurrence of a given variable by its last argument. Their definition on non$-\top^{\mathcal{C}}$ elements is quite standard and we do not present it here. They are extended to $\mathcal{C}$ as follows: $occ(\top^{\mathcal{C}}) \stackrel{\text{def}}{=} \emptyset$, $subst(e, V, \top^{\mathcal{C}})$ equals $e$ when $V \notin occ(e)$ and $\top^{\mathcal{C}}$ when $V \in occ(e)$.

*Abstract Symbolic Environments.* The *symbolic constant domain* is the set $\mathcal{D}^{\mathcal{C}} \stackrel{\text{def}}{=} \mathcal{V} \to \mathcal{C}$ restricted as follows: there must be no cyclic dependencies in a map $S^{\mathcal{C}} \in \mathcal{D}^{\mathcal{C}}$, that is, pair-wise distinct variables $V_1, \ldots, V_n$ such that $\forall i,\ V_i \in occ(S^{\mathcal{C}}(V_{i+1}))$ and $V_n \in occ(S^{\mathcal{C}}(V_1))$. The partial order $\sqsubseteq^{\mathcal{C}}$ on $\mathcal{D}^{\mathcal{C}}$ is the point-wise extension of that on $\mathcal{C}$. Each element $S^{\mathcal{C}} \in \mathcal{D}^{\mathcal{C}}$ represents the set of environments compatible with the symbolic information:

**Definition 6** $\gamma^{\mathcal{C}}(S^{\mathcal{C}}) \stackrel{\text{def}}{=} \{\ \rho \in (\mathcal{V} \to \mathbb{I}) \mid \forall k,\ \rho(V_k) \in [\![ S^{\mathcal{C}}(V_k) ]\!](\rho)\ \}$.

*Main Theorem.* Our approach relies on the fact that applying a substitution from $S^{\mathcal{C}}$ to any expression is sound with respect to $\gamma^{\mathcal{C}}(S^{\mathcal{C}})$:

**Theorem 5** $\forall e, V, S^{\mathcal{C}},\ \gamma^{\mathcal{C}}(S^{\mathcal{C}}) \models e \preceq subst(e, V, S^{\mathcal{C}}(V))$.

*Abstract Operators.* We now define the following operators on $\mathcal{D}^{\mathcal{C}}$:

**Definition 7**

- $\{\!|\, V \leftarrow e \,|\!\}^{\mathcal{C}}(S^{\mathcal{C}})(V_k) \stackrel{\text{def}}{=} \begin{cases} subst(e,\ V,\ S^{\mathcal{C}}(V)) & \text{if } V = V_k \\ subst(S^{\mathcal{C}}(V_k),\ V,\ S^{\mathcal{C}}(V)) & \text{if } V \neq V_k \end{cases}$
- $\{\!|\, e \bowtie 0\ ?\, |\!\}^{\mathcal{C}}(S^{\mathcal{C}}) \stackrel{\text{def}}{=} S^{\mathcal{C}}$,
- $(S^{\mathcal{C}} \cup^{\mathcal{C}} T^{\mathcal{C}})(V_k) \stackrel{\text{def}}{=} \begin{cases} S^{\mathcal{C}}(V_k) & \text{if } S^{\mathcal{C}}(V_k) = T^{\mathcal{C}}(V_k) \\ \top^{\mathcal{C}} & \text{otherwise} \end{cases}$
- $S^{\mathcal{C}} \cap T^{\mathcal{C}} \stackrel{\text{def}}{=} S^{\mathcal{C}}$

Our assignment $V \leftarrow e$ first substitutes $V$ with $S^{\mathcal{C}}(V)$ in $S^{\mathcal{C}}$ and $e$ before adding the information that $V$ maps to the substituted $e$. This is necessary to remove all prior information on $V$ (no longer valid after the assignment) and prevent the apparition of dependency cycles. As we are only interested in propagating assignments, tests are abstracted as the identity, which is sound but coarse. Our

union abstraction only keeps syntactically equal expressions. This corresponds to the least upper bound with respect to $\sqsubseteq^{\mathcal{C}}$. Our intersection keeps only the information of the left argument. All these operators respect the non-cyclicity condition. Note that one could be tempted to refine the intersection by mixing information from the left and right arguments in order to minimize the number of variables mapping to $\top^{\mathcal{C}}$. Unfortunately, careless mixing may break the non-cyclicity condition. We settled, as a simpler but safe solution, to keeping the left argument. Finally, we do not need any widening: at each abstract iteration, unstable symbolic expressions are directly replaced with $\top^{\mathcal{C}}$ when applying $\cup^{\mathcal{C}}$, and so, become stable.

### 5.2 Integration With a Numerical Abstract Domain

Given a numerical abstract domain $\mathcal{D}^\sharp$, the domain $\mathcal{D}^{\sharp\times\mathcal{C}}$ is obtained by combining $\mathcal{D}^\sharp_{\mathcal{L}}$ with $\mathcal{D}^{\mathcal{C}}$ the following way:

**Definition 8**

- $\mathcal{D}^{\sharp\times\mathcal{C}} \stackrel{\text{def}}{=} \mathcal{D}^\sharp \times \mathcal{D}^{\mathcal{C}}$,
- $\sqsubseteq^{\sharp\times\mathcal{C}}$, $\cup^{\sharp\times\mathcal{C}}$ *and* $\cap^{\sharp\times\mathcal{C}}$ *are defined pair-wise, and* $\triangledown^{\sharp\times\mathcal{C}} \stackrel{\text{def}}{=} \triangledown^\sharp \times \cup^{\mathcal{C}}$,
- $\gamma^{\sharp\times\mathcal{C}}(R^\sharp, S^{\mathcal{C}}) \stackrel{\text{def}}{=} \gamma^\sharp(R^\sharp) \cap \gamma^{\mathcal{C}}(S^{\mathcal{C}})$,
- $\{\!|\, V \leftarrow e \,|\!\}^{\sharp\times\mathcal{C}}(R^\sharp, S^{\mathcal{C}}) \stackrel{\text{def}}{=} (\{\!|\, V \leftarrow \mathit{strat}(e, S^{\mathcal{C}}) \,|\!\}^\sharp_{\mathcal{L}}(R^\sharp),\ \{\!|\, V \leftarrow e \,|\!\}^{\mathcal{C}}(S^{\mathcal{C}}))$
- $\{\!|\, e \bowtie 0 \,?\, |\!\}^{\sharp\times\mathcal{C}}(R^\sharp, S^{\mathcal{C}}) \stackrel{\text{def}}{=} (\{\!|\, \mathit{strat}(e, S^{\mathcal{C}}) \bowtie 0 \,?\, |\!\}^\sharp_{\mathcal{L}}(R^\sharp),\ \{\!|\, e \bowtie 0 \,?\, |\!\}^{\mathcal{C}}(S^{\mathcal{C}}))$

*Where* $\mathit{strat}(e, S^{\mathcal{C}})$ *is a* substitution strategy *that may perform sequences of substitutions of the form* $f \mapsto \mathit{subst}(f, V, S^{\mathcal{C}}(V))$ *in* $e$*, for any variables* $V$*.*

All information in $\mathcal{D}^{\mathcal{C}}$ and $\mathcal{D}^\sharp$ are computed independently, except that the symbolic information is used in the transfer functions for $\mathcal{D}^\sharp_{\mathcal{L}}$. The next section discusses the choice of a strategy *strat*. Note that, although we chose in this presentation to abstract the semantics of Fig. 4, our construction can be used on any class of expressions, including floating-point and non-numerical expressions.

### 5.3 Substitution Strategies

Any sequence of substitutions extracted from the current symbolic constant information is sound, but some give better results than others. As for the intervalization of Sect. 4.3, we rely on carefully designed strategies.

*Full Propagation.* Thanks to the non-cyclicity of elements $S^{\mathcal{C}} \in \mathcal{D}^{\mathcal{C}}$, we can safely perform all substitutions $f \mapsto \mathit{subst}(f, V, S^{\mathcal{C}}(V))$ for all $V$ in any order, and reach a normal form. This gives a first basic substitution strategy. However, because our goal is to perform linearization-driven simplifications, it is important to avoid substituting with variable-free expressions or we may lose correlations between multiple occurrences of variables. For instance, full substitution in the assignment $Z \leftarrow X - 0.5 \times Y$ with the environment $S^{\mathcal{C}} = [X \mapsto [0,1],\ Y \mapsto X]$ results in $Z \leftarrow [0,1] - 0.5 \times [0,1]$, and so, $Z \in [-0.5, 1]$. Avoiding variable-free substitutions, this gives $Z \leftarrow X - 0.5 \times X$, and so, $Z \in [0, 0.5]$, which is more

precise. This refined strategy also succeeds in proving that $Y \in [0, 20]$ in the example of Fig. 1 by substituting $Y$ with $X$ in the test $Y \leq 0$.

*Enforcing Determinism and Linearity.* Non-determinism in expressions is a major source of precision loss. Thus, a strategy is to avoid substituting $V$ with $S^{\mathcal{C}}(V)$ whenever $\#(\llbracket S^{\mathcal{C}}(V) \rrbracket \circ \gamma)(X^\sharp) > 1$. As this property is not easily computed, we propose the following sufficient syntactic criterion: $S^{\mathcal{C}}(V)$ should not be $\top^{\mathcal{C}}$ nor contain a non-singleton interval. This strategy gives the expected result in the example of Fig. 1. Likewise, one may wish to avoid substituting with non-linear expressions, as they must be subsequently intervalized, which is a cause of precision loss. However, disabling too many substitutions may prevent the linearization step to exploit correlations. Suppose that we break the last assignment of Fig. 2 in three parts: $U \leftarrow X \times Y$; $V \leftarrow (1 - X) \times Z$; $T \leftarrow U - V$. Then, the interval domain with linearization and symbolic constant propagation will not be able to prove that $T \in [0, 0.3]$ unless we allow substituting, in $T$, $U$ and $V$ with their *non-linear* symbolic value.

*Gaining More Precision.* More precision can be achieved by slightly altering the definition of $\mathcal{D}^{\sharp \times \mathcal{C}}$. A simple but effective idea is to allow several strategies, compute several transfer functions in $\mathcal{D}^\sharp$ in parallel, and take the abstract intersection $\cap^\sharp$ of the results. Another idea is to perform reductions from $\mathcal{D}^{\mathcal{C}}$ to $\mathcal{D}^\sharp$ after each transfer function: $X^\sharp$ is replaced with $\{\!\!\{ V_k - S^{\mathcal{C}}(V_k) = 0 \ ? \}\!\!\}^\sharp(X^\sharp)$ for some $k$. Reductions can be iterated to increase the precision, following Granger's local iterations scheme [10].

## 6 Application to the Astrée Analyzer

Astrée is an efficient static analyzer focusing on the detection of run-time errors for programs written in a subset of the C programming language, excluding recursion, dynamic memory allocation and concurrent executions. It aims towards a degree of precision sufficient to actually *prove* the absence of run-time errors. This is achieved by specializing the analyzer towards specific program families, introducing various abstract domains, and setting iteration strategy parameters. Currently, the considered family of programs is that of safety, critical, embedded, fly-by-wire avionic software, featuring large reactive loops running for billions of iterations, thousands of global state variables, and pervasive floating-point arithmetics. We refer the reader to [1] for more detailed informations on Astrée.

*Integrating the Symbolic Methods.* Astrée uses a partially reduced product of several numerical abstract domains, together with both our two symbolic enhancement methods. Relational domains, such as the octagon [12] or digital filtering [9] domains, rely on the linearization to abstract complex floating-point expressions into interval affine forms on reals. The interval domain is refined by combining three versions of each transfer function. Firstly, using the expression unchanged. Secondly, using the linearized expression. Thirdly, applying symbolic constant propagation followed by linearization. We use the simplification-driven multiplication strategy, as well as the full propagation strategy—not propagating

variable-free expressions.

*Experimental Results.* We present analysis results on a several programs. All the analyses have been carried on an 64-bit AMD Opteron 248 (2 GHz) workstation running Linux, using a single processor. The following table compares the precision and efficiency of ASTRÉE before and after enabling our two symbolic methods:

| | **without enhancements** | | | | **with enhancements** | | | |
|---|---|---|---|---|---|---|---|---|
| code size in lines | analysis time | nb. of iters. | memory | alarms | analysis time | nb. of iters. | memory | alarms |
| 370 | 1.8s | 17 | 16 MB | 0 | 3.1s | 17 | 16 MB | 0 |
| 9 500 | 90s | 39 | 80 MB | 8 | 160s | 39 | 81 MB | 8 |
| 70 000 | 2h 40mn | 141 | 559 MB | 391 | 1h 16mn | 44 | 582 MB | **0** |
| 226 000 | 11h 16mn | 150 | 1.3 GB | 141 | 6h 36mn | 86 | 1.3 GB | 1 |
| 400 000 | 22h 9mn | 172 | 2.2 GB | 282 | 13h 52mn | 96 | 2.2 GB | **0** |

The precision gain is quite impressive as up to hundreds of alarms are removed. In two cases, this increase in precision is sufficient to achieve zero alarm, which actually *proves* the absence of run-time errors. Moreover, the increase in memory consumption is negligible. Finally, in our largest examples, our enhancement methods *save* analysis time: although each abstract iteration is more costly (up to 25%) this is compensated by the reduced number of iterations needed to stabilize our invariants as a smaller state space is explored.

*Discussion.* It is possible to use the symbolic constant propagation also in relational domains, but this was not needed in our examples to remove alarms. Our experiments show that, even though the linearization and constant propagation techniques on intervals are not as robust as fully relational abstract domains, they are quite versatile thanks to their parametrization in terms of strategies, and much simpler to implement than even a simple relational abstract domain. Moreover, our methods exhibit a near-linear time and memory cost, which is much more efficient than relational domains.

## 7 Conclusion

We have proposed, in this article, two techniques, called linearization and symbolic constant propagation, that can be combined together to improve the precision of numerical abstract domains. In particular, we are able to compensate for the lack of non-linear transfer functions in the polyhedron and octagon domains, and for a weak or inexistent level of relationality in the octagon and interval domains. Finally, they help making abstract domains robust against program transformations. Thanks to their parameterization in terms of strategies, they can be finely tuned to take into account semantics as well as syntactic program features. They are also very lightweight in terms of both analysis and development costs. We found out that, in many cases, it is easier and faster to design a couple of linearization and symbolic propagation strategies to solve a local loss of precision in some program, while keeping the interval abstract domain, than to

develop a specific relational abstract domain able to represent the required local properties. Strategies also proved reusable on programs belonging to the same family. Practical results obtained within the ASTRÉE static analyzer show that our methods both increase the precision and save analysis time. They were key in proving the absence of run-time errors in real-life critical embedded avionics software.

*Future Work.* Because the precision gain strongly depends upon the multiplication strategy used in our linearization and the propagation strategy used in the symbolic constant domain, a natural extension of our work is to try and design new such strategies, adapted to different *practical* cases. A more challenging task would be to provide *theoretical* guarantees that some strategies make abstract domains immune to given classes of program transformations.

*Acknowledgments.* We would like to thank all the former and present members of the ASTRÉE team: B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, D. Monniaux and X. Rival. We would also like to thank the anonymous referees for their useful comments.

## References

[1] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *ACM PLDI'03*, volume 548030, pages 196–207. ACM Press, 2003.

[2] F. Bourdoncle. Efficient chaotic iteration strategies with widenings. In *FMPA'93*, volume 735 of *LNCS*, pages 128–14. Springer, 1993.

[3] R. Clarisó and J. Cortadella. The octahedron abstract domain. In *SAS'04*, volume 3148 of *LNCS*, pages 312–327. Springer, 2004.

[4] C. Colby. *Semantics-Based Program Analysis via Symbolic Composition of Transfer Relations.* PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA, 1996.

[5] P. Cousot and R. Cousot. Static determination of dynamic properties of programs. In *ISOP'76*, pages 106–130. Dunod, Paris, France, 1976.

[6] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *ACM POPL'77*, pages 238–252. ACM Press, 1977.

[7] P. Cousot and R. Cousot. Abstract interpretation and application to logic programs. *Journal of Logic Programming*, 13(2–3):103–179, 1992.

[8] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *ACM POPL'78*, pages 84–97. ACM Press, 1978.

[9] J. Feret. Static analysis of digital filters. In *ESOP'04*, volume 2986 of *LNCS*. Springer, 2004.

[10] P. Granger. Improving the results of static analyses programs by local decreasing iteration. In *FSTTCS*, volume 652 of *LNCS*, pages 68–79. Springer, 1992.

[11] G. Kildall. A unified approach to global program optimization. In *ACM POPL'73*, pages 194–206. ACM Press, 1973.

[12] A. Miné. The octagon abstract domain. In *AST 2001 in WCRE 2001*, IEEE, pages 310–319. IEEE CS Press, 2001.

[13] A. Miné. Relational abstract domains for the detection of floating-point run-time errors. In *ESOP'04*, volume 2986 of *LNCS*, pages 3–17. Springer, 2004.

[14] A. Miné. *Weakly Relational Numerical Abstract Domains.* PhD thesis, École Polytechnique, Palaiseau, France, dec 2004.
[15] A. Simon, A. King, and J. Howe. Two variables per linear inequality as an abstract domain. In *LOPSTR'02*, volume 2664 of *LNCS*, pages 71–89. Springer, 2002.
[16] M. Vinícius, A. Andrade, J. L. D. Comba, and J. Stolfi. Affine arithmetic. In *INTERVAL'94*, 1994.