Static Analysis of Numerical Algorithms

Eric Goubault and Sylvie Putot

CEA Saclay, F91191 Gif-sur-Yvette Cedex, France
{eric.goubault, sylvie.putot}@cea.fr

Abstract. We present a new numerical abstract domain for static analysis of the errors introduced by the approximation by floating-point arithmetic of real numbers computation, by abstract interpretation [3]. This work extends a former domain [4,8], with an implicitly relational domain for the approximation of the floating-point values of variables, based on affine arithmetic [2]. It allows us to analyze non trivial numerical computations, that no other abstract domain we know of can analyze with such precise results, such as linear recursive filters of different orders, Newton methods for solving non-linear equations, polynomial iterations, conjugate gradient algorithms.

1 Introduction

The idea of the domain of [4,8][1] is to provide some information on the source of numerical errors in the program. The origin of the main losses of precision is most of the time very localized, so identifying the operations responsible for these main losses, while bounding the total error, can be very useful. The analysis follows the floating-point computation, and bounds at each operation the error committed between the floating-point and the real result. It relies on a model of the difference between the result x of a computation in real numbers, and the result f^x of the same computation using floating-point numbers, expressed as

$$x = f^x + \sum_{\ell \in L \cup \{hi\}} \omega_\ell^x \boldsymbol{\varphi_\ell} \, . \tag{1}$$

In this relation, a term $\omega_\ell^x \boldsymbol{\varphi}_\ell$, $\ell \in L$ denotes the contribution to the global error of the first-order error introduced by the operation labeled ℓ. The value of the error $\omega_\ell^x \in \mathbb{R}$ expresses the rounding error committed at label ℓ, and its propagation during further computations on variable x. Variable $\boldsymbol{\varphi}_\ell$ is a formal variable, associated to point ℓ, and with value 1. Errors of order higher than one, coming from non-affine operations, are grouped in one term associated to special label hi. We refer the reader to [4,8] for the interpretation of arithmetic operations on this domain.

A natural abstraction of the coefficients in expression (1), is obtained using intervals. The machine number f^x is abstracted by an interval of floating-point

[1] Some notations are slightly different from those used in these papers, in order to avoid confusion with the usual notations of affine arithmetic.

K. Yi (Ed.): SAS 2006, LNCS 4134, pp. 18–34, 2006.

numbers, each bound rounded to the nearest value in the type of variable x. The error terms $\omega_i^x \in \mathbb{R}$ are abstracted by intervals of higher-precision numbers, with outward rounding. However, results with this abstraction suffer from the overestimation problem of interval methods. If the arguments of an operation are correlated, the interval computed with interval arithmetic may be significantly wider than the actual range of the result.

Resembling forms, though used in a very different way, were introduced in the interval community, under the name of affine arithmetic [2], to overcome the problem of loss of correlation between variables in interval arithmetic. We propose here a new relational domain, relying on affine arithmetic for the computation of the floating-point value f^x. Indeed, we cannot hope for a satisfying computation of the bounds of the error without an accurate computation of the value, even with very accurate domains for the errors. But affine arithmetic is designed for the estimation of the result of a computation in real numbers. We will show that it is tricky to accurately estimate from there the floating-point result, and that the domain for computing f^x had to be carefully designed.

In section 2, we introduce this new domain and establish the definition of arithmetic operations over it. First ideas on these relational semantics were proposed in [12,13]. In section 3, we present a computable abstraction of this domain, including join and meet operations, and a short insight into practical aspects, such as fixed-point computations, cost of the analysis, and comparison to other domains such as polyhedra. For lack of space, we only give hints of proofs of the correctness of the abstract semantics, in sections 2.3 and 3.1. Finally, we present in section 4, results obtained with the implementation of this domain in our static analyzer FLUCTUAT, that demonstrate its interest.

Notations: Let $\mathbb{F}$ be the set of IEEE754 floating-point numbers (with their infinities), $\mathbb{R}$ the set of real numbers with ∞ and $-\infty$. Let $\uparrow_\circ : \mathbb{R} \to \mathbb{F}$ be the function that returns the rounded value of a real number x, with respect to the rounding mode $\circ$. The function $\downarrow_\circ : \mathbb{R} \to \mathbb{F}$ that returns the roundoff error is defined by

$$\forall x \in \mathbb{R},\ \downarrow_\circ (x) = x - \uparrow_\circ (x).$$

We note $\mathbb{IR}$ the set of intervals with bounds in $\mathbb{R}$. In the following, an interval will be noted in bold, $\boldsymbol{a}$, and its lower and upper bounds will be noted respectively $\underline{a}$ and $\overline{a}$. And we identify when necessary, a number with the interval with its two bounds equal to this number. $\wp(X)$ denotes the set of subsets of X.

2 New Domain for the Floating-Point Value f^x

Affine arithmetic was proposed by De Figueiredo and Stolfi [2], as a solution to the overestimation in interval arithmetic. It relies on forms that allow to keep track of affine correlations between quantities. Noise symbols are used to express the uncertainty in the value of a variable, when only a range is known. The sharing of noise symbols between variables expresses dependencies. We present here a domain using affine arithmetic for the floating-point computation.

In section 2.1, we present briefly the principles of affine arithmetic for real numbers computations. Then in section 2.2, we show on an example the challenges of its adaptation to the estimation of floating-point computations. In sections 2.3 and 2.4, we present the solution we propose, and finally in section 2.5 we demonstrate this solution on the example introduced in section 2.2.

2.1 Affine Arithmetic for Computation in Real Numbers

In affine arithmetic, a quantity x is represented by an affine form, which is a polynomial of degree one in a set of noise terms ε_i :

$$\hat{x} = \alpha_0^x + \alpha_1^x \varepsilon_1 + \ldots + \alpha_n^x \varepsilon_n, \quad \text{with } \varepsilon_i \in [-1, 1] \text{ and } \alpha_i^x \in \mathbb{R}.$$

Let $\mathbb{AR}$ denote the set of such affine forms. Each noise symbol ε_i stands for an independent component of the total uncertainty on the quantity x, its value is unknown but bounded in [-1,1]; the corresponding coefficient α_i^x is a known real value, which gives the magnitude of that component. The idea is that the same noise symbol can be shared by several quantities, indicating correlations among them. These noise symbols can be used not only for modelling uncertainty in data or parameters, but also uncertainty coming from computation.

Let $\mathcal{E}_0$ be the set of expressions on a given set $\mathcal{V}$ of variables (all possible program variables) and constants (intervals of reals), built with operators $+$, $-$, $*$, $/$ and $\sqrt{}$. We note $\hat{\mathcal{C}}_{\mathbb{A}}$ the set of abstract contexts in $\mathbb{AR}$. We can now define, inductively on the syntax of expressions, the evaluation function $\hat{eval} : \mathcal{E}_0 \times \hat{\mathcal{C}}_{\mathbb{A}} \to \mathbb{AR}$. For lack of space, we only deal with a few operations. The assignment of a variable x whose value is given in a range $[a, b]$, introduces a noise symbol ε_i :

$$\hat{x} = (a + b)/2 + (b - a)/2\, \varepsilon_i.$$

The result of linear operations on affine forms, applying polynomial arithmetic, can easily be interpreted as an affine form. For example, for two affine forms $\hat{x}$ and $\hat{y}$, and a real number r, we get

$$\hat{x} + \hat{y} = (\alpha_0^x + \alpha_0^y) + (\alpha_1^x + \alpha_1^y)\varepsilon_1 + \ldots + (\alpha_n^x + \alpha_n^y)\varepsilon_n$$
$$\hat{x} + r = (\alpha_0^x + r) + \alpha_1^x \varepsilon_1 + \ldots + \alpha_n^x \varepsilon_n$$
$$r\hat{x} = r\alpha_0^x + r\alpha_1^x \varepsilon_1 + \ldots + r\alpha_n^x \varepsilon_n$$

For non affine operations, the result applying polynomial arithmetic is not an affine form : we select an approximate linear resulting form, and bounds for the approximation error committed using this approximate form are computed, that create a new noise term added to the linear form. For example, for the multiplication of $\hat{x}$ and $\hat{y}$, defined on the set of noise symbols $\varepsilon_1, \ldots, \varepsilon_n$, a first over-approximation for the result (the one given in [2]), writes

$$\hat{x} \times \hat{y} = \alpha_0^x \alpha_0^y + \sum_{i=1}^{n} (\alpha_i^x \alpha_0^y + \alpha_i^y \alpha_0^x)\varepsilon_i + (\sum_{i=1}^{n} |\alpha_i^x| . |\sum_{i=1}^{n} |\alpha_i^y|) \varepsilon_{n+1}.$$

However, this new noise term can be a large over-estimation of the non-affine part, the additional term is more accurately approximated by

$$\sum_{i=1}^{n} |\alpha_i^x \alpha_i^y| [0,1] + \sum_{1 \le i \ne j \le n} |\alpha_i^x \alpha_j^y| [-1,1].$$

This term is not centered on zero, the corresponding affine form then writes

$$\hat{x} \times \hat{y} = (\alpha_0^x \alpha_0^y + \frac{1}{2} \sum_{i=1}^{n} |\alpha_i^x \alpha_i^y|) + \sum_{i=1}^{n} (\alpha_i^x \alpha_0^y + \alpha_i^y \alpha_0^x) \varepsilon_i + (\frac{1}{2} \sum_{i=1}^{n} |\alpha_i^x \alpha_i^y| + \sum_{i \ne j} |\alpha_i^x \alpha_j^y|) \varepsilon_{n+1}.$$

For example, if $\hat{x} = \varepsilon_1 + \varepsilon_2$ and $\hat{y} = \varepsilon_2$, we get with the first formulation, $\hat{x} \times \hat{y} = 2\varepsilon_3 \in [-2, 2]$ and with the second formulation, $\hat{x} \times \hat{y} = \frac{1}{2} + \frac{3}{2}\varepsilon_3 \in [-1, 2]$. However, the exact range here is [-0.25,2] : indeed there could be a more accurate computation for the multiplication, using Semi-Definite Programming[2].

2.2 Motivation for the Affine Real Form Plus Error Term Domain

Using affine arithmetic for the estimation of floating-point values needs some adaptation. Indeed, the correlations that are true on real numbers after an arithmetic operation, are not exactly true on floating-point numbers.

Consider for example two independent variables x and y that both take their value in the interval [0,2], and the arithmetic expression $((x + y) - y) - x$. Using affine arithmetic in the classical way, we write $x = 1 + \varepsilon_1$, $y = 1 + \varepsilon_2$, and we get zero as result of the expression. This is the expected result, provided this expression is computed in real numbers. But if we take x as the nearest floating-point value to 0.1, and $y = 2$, then the floating-point result is $-9.685755e - 8$.

In order to model the floating-point computation, a rounding error must thus be added to the affine form resulting from each arithmetic operation. But we show here on an example that the natural extensions of real affine arithmetic are not fully satisfying. We consider an iterated computation $x = x - a * x$, for $0 \le a < 1$ and starting with $x_0 \in [0, 2]$.

- With interval arithmetic, $x_1 = x_0 - ax_0 = [-2a, 2]$, and iterating we get an over-approximation (due to the use of floating-point numbers), of the already unsatisfying interval $\boldsymbol{x_n} = [(1-a)^n - (1+a)^n, (1-a)^n + (1+a)^n]$.

- We now consider affine arithmetic with an extra rounding error added for each arithmetic operation. We suppose for simplicity's sake that all coefficients are exactly represented, and we unfold the iterations of the loop. We note u the value $ulp(1)$, which is the absolute value of the difference between 1 and the nearest floating-point number, and $\boldsymbol{u} = [-u, u]$. We note $\hat{f}_n = \hat{x}_n + \boldsymbol{\delta_n}$, where $\hat{x}_n$ is the affine form representing the result of the computation of x_n in real numbers, and $\boldsymbol{\delta_n}$ the interval error term giving the floating-point number. We have $\hat{x}_0 = 1 + \varepsilon_1$ and, using affine arithmetic on real numbers, we get

$$\hat{x}_n = (1-a)^n + (1-a)^n \varepsilon_1 \,, \quad \forall n \ge 0.$$

[2] We thank Stéphane Gaubert who pointed out this to us.

The rounding error on x_0 is $\boldsymbol{\delta_0} = 0$. Using interval arithmetic for the propagation of the error $\boldsymbol{\delta_n}$, and adding the rounding errors corresponding to the product ax_n and to the subtraction $x_n - ax_n$, we get

$$\boldsymbol{\delta_{n+1}} = (1+a)\boldsymbol{\delta_n} + a(1-a)^n\boldsymbol{u} + (1-a)^{n+1}\boldsymbol{u} = (1+a)\boldsymbol{\delta_n} + (1-a)^n\boldsymbol{u} \quad (2)$$

In this computation, $\boldsymbol{u}$ denotes an unknown value in an interval, that can be different at each occurrence of $\boldsymbol{u}$. Using property $(1-a)^n \geq 1 - an, \quad \forall a \in [0,1]$ and $n \geq 1$, we can easily prove that for all n, $n\boldsymbol{u} \subset \boldsymbol{\delta_n}$. The error term increases, and $\hat{f}_n$ is not bounded independently of the iterate n.

- Now, to take into account the dependencies also between the rounding errors, we introduce new noise symbols. For a lighter presentation, these symbols are created after the rounding errors of both multiplication ax and subtraction $x - ax$, and not after each of them. Also, a new symbol is introduced at each iteration, but it agglomerates both new and older errors. In the general case, it will be necessary to keep as many symbols as iterations, each corresponding to a new error introduced at a given iteration. The error term is now computed as an affine form $\hat{\delta}_n = \mu_n\varepsilon_{2,n}$, with $\mu_0 = 0$ and

$$\hat{\delta}_{n+1} = (1-a)\mu_n\varepsilon_{2,n} + a(1-a)^n\boldsymbol{u} + (1-a)^{n+1}\boldsymbol{u}.$$

Introducing a new symbol $\varepsilon_{2,n+1} \in [-1,1]$, it is easy to prove that we can write

$$\hat{\delta}_n = n(1-a)^{n-1}u\,\varepsilon_{2,n} \quad \forall n \geq 1.$$

The error converges towards zero. However, we still loose the obvious information that x_n is always positive. Also the computation can be costly : in the general case, one symbol per operation and iteration of the loop may be necessary.

We now propose a semantics that avoids the cost of extra noise symbols, and with which we will show in section 2.5, that we can prove that $x_n \geq 0$, $\forall n$.

2.3 Semantics for the Floating-Point Value: Abstract Domain

Linear correlations between variables can be used directly on the errors or on the real values of variables, but not on floating-point values. We thus propose to decompose the floating-point value f^x of a variable x resulting from a trace of operations, in the real value of this trace of operations r^x, plus the sum of errors δ^x accumulated along the computation, $f^x = r^x + \delta^x$. Other proposals have been made to overcome this problem, most notably [10].

We present in this section an abstract domain, in the sense that we model a program for sets of inputs and parameters (given in intervals). However, it is not fully computable, as we still consider coefficients of the affine forms to be real numbers. A more abstract semantics, and lattice operations, will be briefly presented in the implementation section 3.

Real Part r^x: Affine Arithmetic. We now index a noise symbol ε_i by the label $i \in L$ corresponding to the operation that created the symbol. The representation is sparse, as all operations do not create symbols. In this section and for

more simplicity, we suppose that at most one operation, executed once, is associated to each label. The generalization will be discussed in the implementation section.

The correctness of the semantics, defined by $\hat{eval}$, is as follows. We note $\boldsymbol{r^x}$ the smallest interval including $\hat{r}^x$ and $\mathcal{C}$, the set of concrete contexts, i.e. functions from the variables to $\mathbb{R}$, seen as a subset of $\hat{\mathcal{C}}_{\mathbb{A}}$. We have an obvious concretisation function $conc_{\mathbb{R}} : \hat{\mathcal{C}}_{\mathbb{A}} \rightarrow \wp(\mathcal{C})$, making all possible choices of values for the noise symbols in the affine forms it is composed of. This also defines γ from affine forms to intervals, which cannot directly define a strong enough correctness criterion. Affine forms define *implicit* relations, we must prove that in whatever expression we are using them, the concretisation as interval of this particular expression contains the concrete values that this expression can take[3]. We have to compare $\hat{eval}$ with the evaluation function $eval : \mathcal{E}_0 \times \mathcal{C} \rightarrow \mathbb{R}$ which computes an arithmetic expression in a given (real number) context. Formally, the semantics of arithmetic expressions in $\mathbb{AR}$, given by $\hat{eval}$, is correct because for all $e \in \mathcal{E}_0$, for all $\hat{C} \in \hat{\mathcal{C}}_{\mathbb{A}}$, we have property:

$$\forall C \in conc_{\mathbb{R}}(\hat{C}), eval(e, C) \in \gamma \circ \hat{eval}(e, \hat{C}) \tag{3}$$

Error Term $\boldsymbol{\delta^x}$: Errors on Bounds Combined with Maximum Error. The rounding errors associated to the bounds $\underline{r^x}$ and $\overline{r^x}$ is the only information needed to get bounds for the floating-point results. In the general case, our semantics only gives ranges for these errors : we note $\boldsymbol{\delta^x_-}$ and $\boldsymbol{\delta^x_+}$ the intervals including the errors due to the successive roundings committed on the bounds $\underline{r^x}$ and $\overline{r^x}$. The set of floating-point numbers taken by variable x after the computation then lies in the interval

$$\boldsymbol{f^x} = [\underline{r^x} + \underline{\delta^x_-}, \overline{r^x} + \overline{\delta^x_+}].$$

Note that $\underline{\delta^x_-}$ can be greater for example than $\overline{\delta^x_+}$, so this is not equivalent to $\boldsymbol{f^x} = \boldsymbol{r^x} + \overline{(\boldsymbol{\delta^x_-} \cup \boldsymbol{\delta^x_+})}$.

In affine arithmetic, the bounds of the set resulting from an arithmetic operation $x \diamond y$ are not always got from the bounds of the operands x and y as in interval arithmetic : in this case, the error inside the set of values is also needed. We choose to represent it by an interval $\boldsymbol{\delta^x_M}$ that bounds all possible errors committed on the real numbers in interval $\boldsymbol{r^x}$.

This intuition can be formalized again using abstract interpretation [3]. We define $\mathbb{D} = \mathbb{AR} \times \mathbb{IR}^3$ and $\tilde{\gamma} : \mathbb{D} \rightarrow \wp(\mathbb{R} \times \mathbb{F})$ by:

$$\tilde{\gamma}(d, \boldsymbol{\delta_M}, \boldsymbol{\delta_+}, \boldsymbol{\delta_-}) = \begin{cases} \{(r, f) \in \mathbb{R} \times \mathbb{F} / r \in \gamma(d), f - r \in \boldsymbol{\delta_M}\} \\ \cap \left\{(r, f) \in \mathbb{R} \times \mathbb{F} / f \geq inf\ \gamma(d) + \underline{\boldsymbol{\delta_-}}\right\} \\ \cap \left\{(r, f) \in \mathbb{R} \times \mathbb{F} / f \leq sup\ \gamma(d) + \overline{\boldsymbol{\delta_+}}\right\} \end{cases}$$

The correctness criterion for the abstract semantics $\square^\sharp$ of an operator $\square$ ($\square_{\mathbb{R}}$ in the real numbers, $\square_{\mathbb{F}}$ in the floating-point numbers) is then the classical:

[3] This is reminiscent to observational congruences dating back to the λ-calculus.

$\forall \tilde{d}, \tilde{e} \in \mathbb{D}, \forall r^x, r^y \in \mathbb{R}, \forall f^x, f^y \in \mathbb{F}$ such that $(r^x, f^x) \in \tilde{\gamma}(d)$ and $(r^y, f^y) \in \tilde{\gamma}(e)$,

$$(r^x \square_{\mathbb{R}} r^y, f^x \square_{\mathbb{F}} f^y) \in \tilde{\gamma}(d \square^\sharp e) \tag{4}$$

Now the order[4] on $\mathbb{D}$ is as follows: $(d, \boldsymbol{\delta_M}, \boldsymbol{\delta_+}, \boldsymbol{\delta_-}) \leq_{\mathbb{D}} (d', \boldsymbol{\delta'_M}, \boldsymbol{\delta'_+}, \boldsymbol{\delta'_-})$ if

$$\begin{cases} d \leq_{\mathbb{D}} d' \\ \boldsymbol{\delta_M} \subseteq \boldsymbol{\delta'_M} \\ \left[\min\ \gamma(d) + \underline{\boldsymbol{\delta_-}}, \max\ \gamma(d) + \overline{\boldsymbol{\delta_+}}\right] \subseteq \left[\min\ \gamma(d') + \underline{\boldsymbol{\delta'_-}}, \max\ \gamma(d') + \overline{\boldsymbol{\delta'_+}}\right] \end{cases}$$

2.4 Arithmetic Operations on Floating-Point Numbers

The error on the result of a binary arithmetic operation $x \diamond y$, with $\diamond \in \{+, \times\}$, is defined as the sum of two terms :

$$\boldsymbol{\delta}_{\cdot}^{\boldsymbol{x \diamond y}} = \boldsymbol{\delta}_{\cdot,\boldsymbol{p}}^{\boldsymbol{x \diamond y}} + \boldsymbol{\delta}_{\cdot,\boldsymbol{n}}^{\boldsymbol{x \diamond y}},$$

with $\cdot \in \{-, +, M\}$. The propagated error $\boldsymbol{\delta}_{\cdot,\boldsymbol{p}}^{\boldsymbol{x \diamond y}}$ is computed from the errors on the operands, and $\boldsymbol{\delta}_{\cdot,\boldsymbol{n}}^{\boldsymbol{x \diamond y}}$ expresses the rounding error due to current operation $\diamond$.

Propagation of the Errors on the Operands. The propagation of the maximum error uses the maximum errors on the operands. For computing the errors on the result, we need to compute the values of the noise symbols $\hat{r^x}$ and $\hat{r^y}$ for which the bounds of $\boldsymbol{r^z}$ are obtained. For that, we compute the values of the ε_i that give the bounds of $\boldsymbol{r^z}$, and check if for these values, we are on bounds of $\boldsymbol{r^x}$ and $\boldsymbol{r^y}$.

Let $\boldsymbol{b_i^z}$, for $i \in L$ such that $\alpha_i^z \neq 0$, be the value of ε_i that maximizes $\hat{r^z}$. We have

$$\underline{r^z} = \alpha_0^z - \sum_{i \in L,\ \alpha_i^z \neq 0} \alpha_i^z b_i^z = \alpha_0^z - \sum_{i \in L} |\alpha_i^z|$$

$$\overline{r^z} = \alpha_0^z + \sum_{i \in L,\ \alpha_i^z \neq 0} \alpha_i^z b_i^z = \alpha_0^z + \sum_{i \in L} |\alpha_i^z|$$

We can then compute the values of x and y that lead to the bounds of $\boldsymbol{r^z}$ (such that $\underline{r^z} = \hat{r_-^x}(z) \diamond \hat{r_-^y}(z)$ and $\overline{r^z} = \hat{r_+^x}(z) \diamond \hat{r_+^y}(z)$) :

$$\hat{r_-^x}(z) = \alpha_0^x - \sum_{\{i,\ \alpha_i^z \neq 0\}} \alpha_i^x b_i^z + \sum_{\{i,\ \alpha_i^z = 0\}} \alpha_i^x \varepsilon_i$$

$$\hat{r_+^x}(z) = \alpha_0^x + \sum_{\{i,\ \alpha_i^z \neq 0\}} \alpha_i^x b_i^z + \sum_{\{i,\ \alpha_i^z = 0\}} \alpha_i^x \varepsilon_i$$

We note $\boldsymbol{e_-^x(z)}$ (resp $\boldsymbol{e_+^x(z)}$) the interval of error associated to $\hat{r_-^x}(z)$ (resp $\hat{r_+^x}(z)$), used to get the lower bound $\underline{r^z}$ (resp the upper bound $\overline{r^z}$) of the result :

[4] Depending on the order on $\mathbb{AR}$ to be formally defined in section 3.1.

$$\boldsymbol{e}^{\boldsymbol{x}}_{-}(\boldsymbol{z}) = \begin{cases} \boldsymbol{\delta}^{\boldsymbol{x}}_{-} & \text{if } \underline{r^x_-}(z) = \overline{r^x_-}(z) = \underline{r^x}, \\ \boldsymbol{\delta}^{\boldsymbol{x}}_{+} & \text{if } \overline{r^x_-}(z) = \overline{r^x_-}(z) = \overline{r^x}, \\ \boldsymbol{\delta}^{\boldsymbol{x}}_{\boldsymbol{M}} & \text{otherwise.} \end{cases} \quad \boldsymbol{e}^{\boldsymbol{x}}_{+}(\boldsymbol{z}) = \begin{cases} \boldsymbol{\delta}^{\boldsymbol{x}}_{+} & \text{if } \underline{r^x_+}(z) = \overline{r^x_+}(z) = \overline{r^x}, \\ \boldsymbol{\delta}^{\boldsymbol{x}}_{-} & \text{if } \overline{r^x_+}(z) = \overline{r^x_+}(z) = \underline{r^x}, \\ \boldsymbol{\delta}^{\boldsymbol{x}}_{\boldsymbol{M}} & \text{otherwise.} \end{cases}$$

We deduce the following determination of $\boldsymbol{e}^{\boldsymbol{x}}_{-}(\boldsymbol{z})$ and $\boldsymbol{e}^{\boldsymbol{x}}_{+}(\boldsymbol{z})$:
- if $\forall i \in L$ such that $\alpha^x_i \neq 0$, $\alpha^x_i \alpha^z_i > 0$, then $\boldsymbol{e}^{\boldsymbol{x}}_{-}(\boldsymbol{z}) = \boldsymbol{\delta}^{\boldsymbol{x}}_{-}$ and $\boldsymbol{e}^{\boldsymbol{x}}_{+}(\boldsymbol{z}) = \boldsymbol{\delta}^{\boldsymbol{x}}_{+}$
- else if $\forall i \in L$ such that $\alpha^x_i \neq 0$, $\alpha^x_i \alpha^z_i < 0$, then $\boldsymbol{e}^{\boldsymbol{x}}_{-}(\boldsymbol{z}) = \boldsymbol{\delta}^{\boldsymbol{x}}_{+}$ and $\boldsymbol{e}^{\boldsymbol{x}}_{+}(\boldsymbol{z}) = \boldsymbol{\delta}^{\boldsymbol{x}}_{-}$
- else $\boldsymbol{e}^{\boldsymbol{x}}_{-}(\boldsymbol{z}) = \boldsymbol{e}^{\boldsymbol{x}}_{+}(\boldsymbol{z}) = \boldsymbol{\delta}^{\boldsymbol{x}}_{\boldsymbol{M}}$.

Then, using these notations, we can state the propagation rules

$$\begin{aligned} \boldsymbol{\delta}^{\boldsymbol{x+y}}_{\boldsymbol{-,p}} &= \boldsymbol{e}^{\boldsymbol{x}}_{-}(\boldsymbol{x+y}) + \boldsymbol{e}^{\boldsymbol{y}}_{-}(\boldsymbol{x+y}) \\ \boldsymbol{\delta}^{\boldsymbol{x+y}}_{\boldsymbol{+,p}} &= \boldsymbol{e}^{\boldsymbol{x}}_{+}(\boldsymbol{x+y}) + \boldsymbol{e}^{\boldsymbol{y}}_{+}(\boldsymbol{x+y}) \\ \boldsymbol{\delta}^{\boldsymbol{x+y}}_{\boldsymbol{M,p}} &= \boldsymbol{\delta}^{\boldsymbol{x}}_{\boldsymbol{M}} + \boldsymbol{\delta}^{\boldsymbol{y}}_{\boldsymbol{M}} \end{aligned}$$

$$\begin{aligned} \boldsymbol{\delta}^{\boldsymbol{x\times y}}_{\boldsymbol{-,p}} &= \boldsymbol{e}^{\boldsymbol{x}}_{-}(\boldsymbol{x\times y})\underline{r^y(x\times y)} + \boldsymbol{e}^{\boldsymbol{y}}_{-}(\boldsymbol{x\times y})\underline{r^x(x\times y)} + \boldsymbol{e}^{\boldsymbol{x}}_{-}(\boldsymbol{x\times y})\boldsymbol{e}^{\boldsymbol{y}}_{-}(\boldsymbol{x\times y}) \\ \boldsymbol{\delta}^{\boldsymbol{x\times y}}_{\boldsymbol{+,p}} &= \boldsymbol{e}^{\boldsymbol{x}}_{+}(\boldsymbol{x\times y})\overline{r^y(x\times y)} + \boldsymbol{e}^{\boldsymbol{y}}_{+}(\boldsymbol{x\times y})\overline{r^x(x\times y)} + \boldsymbol{e}^{\boldsymbol{x}}_{+}(\boldsymbol{x\times y})\boldsymbol{e}^{\boldsymbol{y}}_{+}(\boldsymbol{x\times y}) \\ \boldsymbol{\delta}^{\boldsymbol{x\times y}}_{\boldsymbol{M,p}} &= \boldsymbol{\delta}^{\boldsymbol{x}}_{\boldsymbol{M}}\boldsymbol{r}^{\boldsymbol{y}} + \boldsymbol{\delta}^{\boldsymbol{y}}_{\boldsymbol{M}}\boldsymbol{r}^{\boldsymbol{x}} + \boldsymbol{\delta}^{\boldsymbol{x}}_{\boldsymbol{M}}\boldsymbol{\delta}^{\boldsymbol{y}}_{\boldsymbol{M}} \end{aligned}$$

Addition of the New Rounding Error. Adding the propagation error to the result of the computation in real numbers, we get the real result of the computation of $f^x \diamond f^y$. We then have to add a new error corresponding to the rounding of this quantity to the nearest floating-point number.

We note $\downarrow_\circ (\boldsymbol{i})$, the possible rounding error on a real number in an interval $\boldsymbol{i}$. We suppose the rounding mode used for the execution is to the nearest floating-point, and note it "n" as subscript.

$$\downarrow_n (\boldsymbol{i}) = \begin{cases} \downarrow_n (\underline{i}) & \text{if } \underline{i} = \overline{i}, \\ \frac{1}{2}\text{ulp}(\max(|\underline{i}|, |\overline{i}|))[-1, 1] & \text{otherwise.} \end{cases}$$

Then, the new rounding error is defined by

$$\begin{aligned} \boldsymbol{\delta}^{\boldsymbol{x\diamond y}}_{\boldsymbol{-,n}} &= -\downarrow_n (\underline{r^{x\diamond y}} + \underline{\delta^{x\diamond y}_{-,p}}) \\ \boldsymbol{\delta}^{\boldsymbol{x\diamond y}}_{\boldsymbol{+,n}} &= -\downarrow_n (\overline{r^{x\diamond y}} + \overline{\delta^{x\diamond y}_{+,p}}) \\ \boldsymbol{\delta}^{\boldsymbol{x\diamond y}}_{\boldsymbol{M,n}} &= -\downarrow_n (\boldsymbol{r}^{\boldsymbol{x\diamond y}} + \boldsymbol{\delta}^{\boldsymbol{x\diamond y}}_{\boldsymbol{M,p}}) \end{aligned}$$

Note that the new rounding errors on the bounds, $\boldsymbol{\delta}^{\boldsymbol{f\diamond g}}_{\boldsymbol{-,n}}$ and $\boldsymbol{\delta}^{\boldsymbol{f\diamond g}}_{\boldsymbol{+,n}}$, are in fact real numbers, identified to a zero-width interval.

These error computations are correct with respect to (4), section 2.3.

2.5 Example

We consider again the example introduced in section 2.2, and we now use the domain just described. The real part is computed using affine arithmetic, as in section 2.2. We have $\boldsymbol{\delta}^{\boldsymbol{x_0}}_{-} = \boldsymbol{\delta}^{\boldsymbol{x_0}}_{+} = \boldsymbol{\delta}^{\boldsymbol{x_0}}_{\boldsymbol{M}} = 0$, and, for n greater or equal than 1,

$$\boldsymbol{\delta}_-^{a\boldsymbol{x}_n} = a\boldsymbol{\delta}_-^{\boldsymbol{x}_n} + \downarrow_n (a\boldsymbol{\delta}_-^{\boldsymbol{x}_n})$$
$$\boldsymbol{\delta}_+^{a\boldsymbol{x}_n} = a\boldsymbol{\delta}_+^{\boldsymbol{x}_n} + \downarrow_n (2a(1-a)^n + a\boldsymbol{\delta}_-^{\boldsymbol{x}_n})$$

Using $\boldsymbol{\delta}_-^{-a\boldsymbol{x}_n} = -\boldsymbol{\delta}_+^{a\boldsymbol{x}_n}$, we deduce

$$\begin{aligned}\boldsymbol{\delta}_-^{\boldsymbol{x}_{n+1}} &= \boldsymbol{\delta}_-^{\boldsymbol{x}_n} + \boldsymbol{\delta}_+^{-a\boldsymbol{x}_n} + \downarrow_n (\boldsymbol{\delta}_-^{\boldsymbol{x}_n} + \boldsymbol{\delta}_+^{-a\boldsymbol{x}_n}) \\ &= (1-a)\boldsymbol{\delta}_-^{\boldsymbol{x}_n} - \downarrow_n (a\boldsymbol{\delta}_-^{\boldsymbol{x}_n}) + \downarrow_n ((1-a)\boldsymbol{\delta}_-^{\boldsymbol{x}_n} - \downarrow_n (a\boldsymbol{\delta}_-^{\boldsymbol{x}_n}))\end{aligned}$$

As $\boldsymbol{\delta}_-^{\boldsymbol{x}_0}$ is zero, the error on the lower bound of $\boldsymbol{x_n}$ stays zero : $\boldsymbol{\delta}_-^{\boldsymbol{x}_n} = 0$ for all n. This means in particular that $f^{x_n} \geq 0$. The same computation for the error on the upper bound leads to

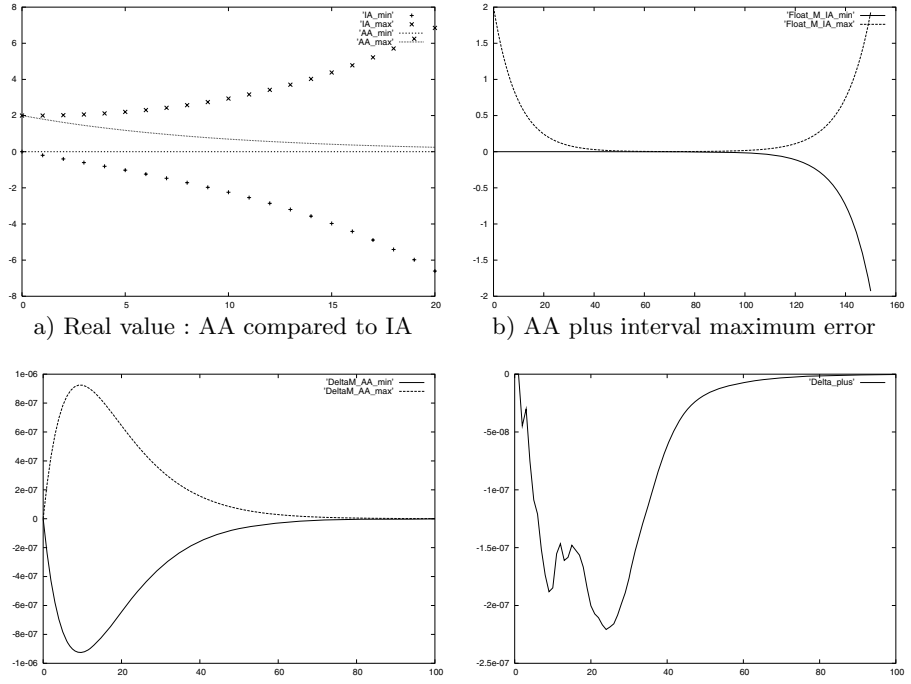
$$\begin{aligned}\boldsymbol{\delta}_+^{\boldsymbol{x}_{n+1}} &= (1-a)\boldsymbol{\delta}_+^{\boldsymbol{x}_n} - \downarrow_n (2a(1-a)^n + a\boldsymbol{\delta}_+^{\boldsymbol{x}_n}) \\ &\quad + \downarrow_n (2(1-a)^{n+1} + (1-a)\boldsymbol{\delta}_+^{\boldsymbol{x}_n} - \downarrow_n (2a(1-a)^n + a\boldsymbol{\delta}_+^{\boldsymbol{x}_n}))\end{aligned}$$

Using real numbers, errors on the lower and upper bounds could be computed exactly. The maximum error on the interval is got by the same computation as in section 2.2 with no extra noise symbols for the errors, that is by (2). Indeed, we could also improve the computation of the maximum error this way, but it will be no longer useful with the (future) relational computation of the errors, to be published elsewhere.

The results got here and in section 2.2, are illustrated in figure 1. In 1 a), the bounds of the computation in real numbers for interval (IA) and affine (AA) arithmetic are compared : the computation by affine arithmetic gives the actual result. In 1 b), we add to the affine arithmetic result the maximum rounding error computed as an interval, and we see that after about 120 iterates, the rounding error prevails and the result diverges. Then in 1 c), we represent the maximum rounding error computed using extra noise symbols. And finally, in 1 d), we represent the rounding error computed on the higher bound of the real interval : it is always negative. Remembering that the error on the lower bound is zero, this proves that the floating-point computation is bounded by the result obtained from the affine computation in real numbers. The fixpoint computation is not presented, as it requires the join operator presented thereafter. However, the analysis does converge to the actual fixpoint.

3 Implementation Within the Static Analyzer FLUCTUAT

We define here a computable abstraction of the domain presented in section 2.3. We now abstract further away from trace semantics : we need control-flow join and meet operators, which must be designed with special care in order to get an efficient analysis of loops. Also, the analyzer does not have access to real numbers, we bound real coefficients by intervals. The semantics for arithmetic operations presented in section 2.3 must thus be extended to interval coefficients. Finally, we insist on the interest of our analysis, in terms of cost and accuracy, compared to existing domains such as polyhedrons.



a) Real value : AA compared to IA
b) AA plus interval maximum error
c) Refined maximum error
d) Error on the higher bound of the real

Fig. 1. Evolution of xn and rounding errors with iterations

3.1 Extended Abstract Domain

We note $\mathbb{AI}$ the set of affine forms $\hat{x} = \alpha_0^x + \boldsymbol{\alpha_1^x}\varepsilon_1 + \ldots + \boldsymbol{\alpha_n^x}\varepsilon_n$ with $\alpha_0^x \in \mathbb{R}$ and $\boldsymbol{\alpha_i^x} \in \mathbb{IR}$ $(i > 0)$. $\mathbb{AR}$ is seen as a subset of $\mathbb{AI}$. Let now $\mathcal{E}$ be the set of expressions on variables in $\mathcal{V}$, constant sets, and built with operators $+$, $-$, $*$, $/$, $\sqrt{}$, $\cup$ and $\cap$. The semantics we are going to define, through $\hat{eval}$ generalized to expressions in $\mathcal{E}$ and for $\mathbb{AI}$, is correct with respect to criterion as (3), but now with expressions in $\mathcal{E}$. We will only need to define the additional join $\cup$ and meet $\cap$ operations.

The set $\mathbb{AI}$ forms a poset, with the following order: $\hat{f} \leq \hat{g}$ if for all variables x, for all abstract contexts $\hat{C}$, calling $\hat{C}_{\hat{f}}$ (respectively $\hat{C}_{\hat{g}}$) the context which has value $\hat{C}(y)$ for all variables $y \neq x$, and value $\hat{f}$ (respectively $\hat{g}$) for variable x, we have:

$$conc_{\mathbb{R}} \circ \hat{eval}(e, \hat{C}_{\hat{f}}) \subseteq conc_{\mathbb{R}} \circ \hat{eval}(e, \hat{C}_{\hat{g}})$$

Note this implies that the concretization as a subset of $\mathbb{R}$ of $\hat{C}_{\hat{f}}$ is included in the concretization as a subset of $\mathbb{R}$ of $\hat{C}_{\hat{g}}$ (take $e = x$). Note as well that this is coherent with property (3), defining correctness: any bigger affine interval than a correct one remains correct. Unfortunately, this does not define a lattice, and we will only have approximate join and meet operations. Also, an important prop-

erty is that $conc_{\mathbb{R}}$ does not always provide with an upper approximation of an environment, i.e. intervals are not always less precise than affine forms, depending on the "continuation". This can be true though, for instance if continuations only contain linear expressions.

3.2 Join and Meet Operations

Affine Forms. Technically, we use a *reduced product* of the domain of affine intervals with the domain of intervals. As we just saw, it is not true that the evaluation of any expression using affine forms is always more accurate than the evaluation of the same expression using intervals (i.e. $\hat{f} \leq \gamma(\hat{f})$).

For any interval $\boldsymbol{i}$, we note

$$\text{mid}(\boldsymbol{i}) = \uparrow_\circ \left(\frac{\underline{i} + \overline{i}}{2} \right), \quad \text{dev}(\boldsymbol{i}) = max(\uparrow_\circ (\overline{i} - \text{mid}(\boldsymbol{i})), \uparrow_\circ (\text{mid}(\boldsymbol{i}) - \underline{i}))$$

the center and deviation of the interval, using finite precision numbers. Suppose for instance $\alpha_0^x \leq \alpha_0^y$. A natural join between affine forms $\hat{r^x}$ and $\hat{r^y}$, associated to a new label k is

$$\hat{r}^{x \cup y} = \text{mid}([\alpha_0^x, \alpha_0^y]) + \sum_{i \in L} (\boldsymbol{\alpha_i^x} \cup \boldsymbol{\alpha_i^y}) \varepsilon_i + \text{dev}([\alpha_0^x, \alpha_0^y]) \varepsilon_k \quad (5)$$

This join operation is an upper bound of $\hat{r^x}$ and $\hat{r^y}$ in the order defined in section 3, but might be greater than the union of the corresponding intervals. However, if the over-approximation is not too large, it is still interesting to keep the relational formulation for further computations.

There is no natural intersection on affine forms, except in particular cases. In the general case, a possibility is to define the meet (at a new label k) of the affine forms as the intersection of the corresponding intervals :

$$\hat{r}^{x \cap y} = \text{mid}(\boldsymbol{r^x} \cap \boldsymbol{r^y}) + \text{dev}(\boldsymbol{r^x} \cap \boldsymbol{r^y}) \varepsilon_k$$

Another simple possibility is to take for $\hat{r}^{x \cap y}$ the smaller of the two affine forms $\hat{r}^x$ and $\hat{r}^y$, in the sense of the width of the concretized intervals $\boldsymbol{r^x}$ and $\boldsymbol{r^y}$.

Also, a relation can sometimes be established between the noise symbols of the two affine forms, that may be used in further computations.

Error Domain. The union on the intervals of possible errors due to successive roundings is

$$\boldsymbol{\delta}_M^{\boldsymbol{x} \cup \boldsymbol{y}} = \boldsymbol{\delta}_M^{\boldsymbol{x}} \cup \boldsymbol{\delta}_M^{\boldsymbol{y}}.$$

For errors on the bounds, a natural and correct union is $\boldsymbol{\delta}_-^{\boldsymbol{x} \cup \boldsymbol{y}} = \boldsymbol{\delta}_-^{\boldsymbol{x}} \cup \boldsymbol{\delta}_-^{\boldsymbol{y}}$ and $\boldsymbol{\delta}_+^{\boldsymbol{x} \cup \boldsymbol{y}} = \boldsymbol{\delta}_+^{\boldsymbol{x}} \cup \boldsymbol{\delta}_+^{\boldsymbol{y}}$. However, the set of floating-point values coming from this model can be largely overestimated in the cases when the union of affine forms gives a larger set of values than $\boldsymbol{r^x} \cup \boldsymbol{r^y}$ would do. We thus propose to use a more accurate model, still correct with respect to correctness criterion (4), where $\boldsymbol{\delta}_-^{\boldsymbol{x} \cup \boldsymbol{y}}$ is no longer the error on the lower bound due to successive roundings, but

the representation error between the minimum value represented by the affine form, and the minimum of the floating-point value (same thing for the error on the maximum bound) :

$$\boldsymbol{\delta}_{-}^{\boldsymbol{x}\cup\boldsymbol{y}} = \left(\boldsymbol{\delta}_{-}^{\boldsymbol{x}} + \underline{r^x} - \underline{r^{x\cup y}}\right) \bigcup \left(\boldsymbol{\delta}_{-}^{\boldsymbol{y}} + \underline{r^y} - \underline{r^{x\cup y}}\right)$$
$$\boldsymbol{\delta}_{+}^{\boldsymbol{x}\cup\boldsymbol{y}} = \left(\boldsymbol{\delta}_{+}^{\boldsymbol{x}} + \overline{r^x} - \overline{r^{x\cup y}}\right) \bigcup \left(\boldsymbol{\delta}_{+}^{\boldsymbol{y}} + \overline{r^y} - \overline{r^{x\cup y}}\right)$$

A disturbing aspect of this model is that we no longer have for all variable x, $\boldsymbol{\delta}_{-}^{\boldsymbol{x}} \subset \boldsymbol{\delta}_{\boldsymbol{M}}^{\boldsymbol{x}}$ and $\boldsymbol{\delta}_{+}^{\boldsymbol{x}} \subset \boldsymbol{\delta}_{\boldsymbol{M}}^{\boldsymbol{x}}$. However, we still have $\underline{\delta_{-}^{x}} \geq \underline{\delta_{M}^{x}}$ and $\overline{\delta_{+}^{x}} \leq \overline{\delta_{M}^{x}}$.

For the meet operation on errors, we define the obvious:

$$\boldsymbol{\delta}_{\boldsymbol{M}}^{\boldsymbol{x}\cap\boldsymbol{y}} = \boldsymbol{\delta}_{\boldsymbol{M}}^{\boldsymbol{x}} \cap \boldsymbol{\delta}_{\boldsymbol{M}}^{\boldsymbol{y}}$$
$$\boldsymbol{\delta}_{-}^{\boldsymbol{x}\cap\boldsymbol{y}} = \boldsymbol{\delta}_{-}^{\boldsymbol{x}} \text{ if } \underline{r^x} \geq \underline{r^y}, \text{ else } \boldsymbol{\delta}_{-}^{\boldsymbol{y}}$$
$$\boldsymbol{\delta}_{+}^{\boldsymbol{x}\cap\boldsymbol{y}} = \boldsymbol{\delta}_{+}^{\boldsymbol{x}} \text{ if } \overline{r^x} \leq \overline{r^y}, \text{ else } \boldsymbol{\delta}_{+}^{\boldsymbol{y}}$$

3.3 Loops and Widening

In practice, a label may correspond not to a unique operation, but to sets of operations (for example a line of program or a function). The semantics can be easily extended to this case, creating noise symbols only when a label is met.

Moreover, in loops, different noise symbols will have to be introduced for the same arithmetic operation at different iterations of the loop : a first solution, accurate but costly, is to introduce each time a new symbol, that is $\varepsilon_{i,k}$ for label i in the loop and iteration k of the analyzer on the loop, and to keep all symbols. A fixpoint is got when the error terms are stable, for each label j introduced out of the loop, the interval coefficient $\boldsymbol{\alpha}_{\boldsymbol{j}}^{\boldsymbol{x_n}}$ is stable, and for each label i introduced in the loop, the sum of contributions $\sum_{k=1}^{n} \boldsymbol{\alpha}_{\boldsymbol{i,k}}^{\boldsymbol{x_n}}[-1,1]$ is stable[5]. That is, a fixpoint of a loop is got at iteration n for variable x if

$$\begin{array}{ll} \boldsymbol{\delta}_{-}^{\boldsymbol{x_n}} \subset \boldsymbol{\delta}_{-}^{\boldsymbol{x_{n-1}}}, \quad \boldsymbol{\delta}_{+}^{\boldsymbol{x_n}} \subset \boldsymbol{\delta}_{+}^{\boldsymbol{x_{n-1}}}, \quad \boldsymbol{\delta}_{\boldsymbol{M}}^{\boldsymbol{x_n}} \subset \boldsymbol{\delta}_{\boldsymbol{M}}^{\boldsymbol{x_{n-1}}} & \\ \boldsymbol{\alpha}_{\boldsymbol{j}}^{\boldsymbol{x_n}} \subset \boldsymbol{\alpha}_{\boldsymbol{j}}^{\boldsymbol{x_{n-1}}} & \text{for all } j \text{ outside the loop} \\ \sum_{k=1}^{n} \boldsymbol{\alpha}_{\boldsymbol{i,k}}^{\boldsymbol{x_n}}[-1,1] \subset \sum_{k=1}^{n-1} \boldsymbol{\alpha}_{\boldsymbol{i,k}}^{\boldsymbol{x_{n-1}}}[-1,1] & \text{for all } i \text{ in the loop} \end{array}$$

In the same way, a natural widening consists in applying a standard widening componentwise on errors, on coefficients of the affine forms for labels outside the loop, and on the sum $\sum_{k=1}^{n} \boldsymbol{\alpha}_{\boldsymbol{i,k}}^{\boldsymbol{x_n}}[-1,1]$ for a label i in the loop. However, in some cases, reducing the affine form, or part of it, to an interval after a number of iterations, allows to get a finite fixpoint while the complete form does not.

Another possible implementation is to keep only dependencies between a limited number of iterations of a loop, and agglomerate older terms introduced in the loop. For example, a first order recurrence will need only dependencies from one iteration to the next to get accurate results, while higher order recur-

[5] This is a correct criterion with respect to the order defined in section 3.1, but weaker conditions may be used as well.

rences will need to keep more information. This problem has to be considered again when getting out of the loop, for a good trade-off between efficiency and accuracy.

3.4 Use of Finite Precision Numbers in the Analysis

The analyzer does not have access to real numbers, real coefficients in the affine forms are abstracted using intervals with outward rounding. We use for this the MPFR library [11] that provides arithmetic on arbitrary precision floating-point numbers, with exact rounding. However, the abstract domain defined in 3.1 has a real and not an interval coefficient α_0^x. Technically, this is achieved by creating a new noise symbol whenever coefficient α_0^x can no longer be computed exactly with the precision used. Morally, these additional noise symbols are used to keep the maximum of correlations, even between errors introduced artificially because of the imprecision of the analysis. Also, in some cases, using high precision numbers is useful to get more accurate results.

3.5 Comparison with Related Abstract Domains

There is a concretisation operator from affine intervals to polyhedra, whose image is the set of center-symmetric bounded polyhedra. Calling m the number of variables, n the number of noise symbols, the joint range of the m variables is a polyhedra with at most of the order of $2n$ faces within a n-dimensional linear subspace of R^m (if $m \geq n$). Conversely, there is no optimal way in general to get an affine form containing a given polyhedra.

Zones [9] are particular center-symmetric bounded polyhedra, intersected with hypercubes, so our domain is more general (since we always keep affine forms together with an interval abstraction), even though less general than polyhedra. It is more comparable to templates [7], where new relations are created along the way, when needed through the evaluation of the semantic functional.

We illustrate this with the following simple program (labels are given as comments):

```
x = [0,2]    // 1      z = xy;       // 3
y = x+[0,2]  // 2      t = z-2*x-y;  // 4
```

In the polyhedral approach, we find as invariants the following ones:

line 2	line 3	line 4
$\begin{cases} 0 \leq x \leq 2 \\ 0 \leq y - x \leq 2 \end{cases}$	$\begin{cases} 0 \leq x \leq 2 \\ 0 \leq y - x \leq 2 \\ 0 \leq z \leq 8 \end{cases}$	$\begin{cases} 0 \leq x \leq 2 \\ 0 \leq y - x \leq 2 \\ 0 \leq z \leq 8 \\ -8 \leq t \leq 8 \end{cases}$

At line 3, we used the concretisation of the invariant of line 2 on intervals to get the bounds for `z`, as is customarily done in zones and polyhedra for non-linear expressions. The particular polyhedra that affine intervals represent make it possible to interpret precisely non-linear expressions, which are badly handled in other linear relational domains:

line 2	line 3	line 4
$\begin{cases} x = 1 + \epsilon_1 \\ y = 2 + \epsilon_1 + \epsilon_2 \end{cases}$	$\begin{cases} x = 1 + \epsilon_1 \\ y = 2 + \epsilon_1 + \epsilon_2 \\ z = \frac{5}{2} + 3\epsilon_1 + \epsilon_2 + \frac{3}{2}\epsilon_3 \in [-3, 8] \end{cases}$	$\begin{cases} x = 1 + \epsilon_1 \\ y = 2 + \epsilon_1 + \epsilon_2 \\ z = \frac{5}{2} + 3\epsilon_1 + \epsilon_2 + \frac{3}{2}\epsilon_3 \\ t = -\frac{3}{2} + \frac{3}{2}\epsilon_3 \in [-3, 0] \end{cases}$

Notice the polyhedral approach is momentarily, at line 3, better than the estimate given by affine arithmetic [6], but the relational form we compute gives much better results in subsequent lines: `t` has in fact exact range in $[-\frac{9}{4}, 0]$ close to what we found: $[-3, 0]$. This is because the representation of z contains implicit relations that may prove useful in further computations, that one cannot guess easily in the explicit polyhedral format (see the work [7] though).

Another interest of the domain is that the implicit formulation of relations is very economical (in time and memory), with respect to explicit formulations, which need closure operators, or expensive formulations (such as with polyhedra). For instance: addition of two affine forms with n noise symbols costs n elementary operations, independently of the number of variables. Multiplication costs n^2 elementary operations. Moreover, affine operations (addition and subtraction) do not introduce new noise symbols, and existing symbols can be easily agglomerated to reduce this number n. This leads to an analysis whose cost can be fairly well controlled.

It is well known that it is difficult to use polyhedra when dealing with more than a few tens or of the order of one hundred variables. We actually used this domain on programs containing of the order of a thousand variables (see example `CG10` where we deal with 189 variables already) with no help from any partitioning technique.

4 Examples

Our static analyzer Fluctuat is used in an industrial context, mostly for validating instrumentation and control code. We refer the reader to [6] for more on our research for industrial applications, but present here some analysis results. They show that the new domain for the values of variables is of course more expensive than interval arithmetic, but comparable to the domain used for the errors. And it allows us to accurately analyze non trivial numerical computations.

Consider the program of figure 2 that computes the inverse of `A` by a Newton method. The assertion `A = __BUILTIN_DAED_DBETWEEN(20.0,30.0)` tells the analyzer that the double precision input `A` takes its value between 20.0 and 30.0. Then the operation `PtrA = (signed int *) (&A)` casts `A` into an array of two integers. Its exponent `exp` is got from the first integer. Thus we have an initial estimate of the inverse, `xi`, with $2^{-\texttt{exp}}$. Then a non linear iteration is computed until the difference `temp` between two successive iterates is bounded by e-10.

[6] However, as pointed out in section 2.1, we could use a more accurate semantics for the multiplication. Note also that in our analyzer, we are maintaining a reduced product between affine forms and intervals, hence we would find here the same enclosure for z as with general polyhedra.

Here, using the relational domain, Fluctuat proves that, for all inputs between 20.0 and 30.0, the algorithm terminates in a number of iterations between 5 and 9, and states that the output `xi` is in the interval [3.33e-2,5.00e-2] with an error due to rounding in [-4.21e-13,4.21e-13]. Executions confirm that respectively 5 iterations for `A = 20.0`, and 9 iterations for `A = 30.0`, are needed. Exact bounds for the number of iterations of this loop for a range of input values is a difficult information to be synthetized by the analyzer : indeed, if we study the same algorithm for simple precision floating-point numbers, instead of double precision, there are cases in which the algorithm does not terminate. Also, the interval for the values indeed is a tight enclosure of the inverse of the inputs. The error is over-estimated, but this will be improved by the future relational domain on the errors. More on this example can be found in [6].

Now, to demontrate the efficiency of our approach, we used it on several typical examples, with performances shown on the table below. Column #l is the number of lines of C code of the program, #v describes the number of variables known to the main function (local variables are not counted). Column Int shows the floating-point value plus global error result, using an interval abstraction of the floating-point value. On the next line is the time spent by the analyzer, in seconds (laptop PC, Pentium M 800MHz, 512Mb of memory), and the maximal memory it had to use (which is allocated by big chunks, hence the round figures). The same is done in column Aff, with the affine forms plus error domain.

Name	#l	#v (fl/int)	Int (time/mem)	Aff (time/mem)
Poly	8	3 (3/0)	[-7,8] + [-3.04,3.04]e-6 ε (0s/4Mb)	[-2.19, 2.75] +[-2.2,2.2]e-6 ε (0.01s/4Mb)
Inv	26	9 (4/5)	[-∞,∞] + [-∞,∞]ε ($\geq$12000s/4Mb)	[3.33,5]e-2 + [-4.2,4.2]e-13 ε (228s/4Mb)
F1a	29	8 (6/2)	[-∞,∞] + [-∞,∞]ε (0.1s/4Mb)	[-10,10] + [-∞,∞]ε (0.63s/7Mb)
F1b idem	11	6 (4/2)	[-∞,∞] + [-∞,∞]ε (0.03/4Mb) [-1.9,1.9]e2 + [-4.8,4.8]e-3 ε (9.66s/8Mb)	[-0.95,0.95] + [-∞,∞]ε (0.26/4Mb)
F2	19	7 (6/1)	[-2.5,2.5]e12 + [-2.3,2.3]e-2 ε (0.13s/4Mb)	[-1.22e-4,1.01] + [-9.4,9.4]e-4 ε (0.45s/7Mb)
SA	164	32 (24/8)	[1.06,2.52] + [-4.4,4.4]e-5ε (24.96s/16Mb)	[1.39,2.03] + [-4.1,4.1]e-5 ε (25.2s/16Mb)
SH	162	9 (7/2)	[-∞,∞] + [-∞,∞]ε (116.72s/4Mb)	[4.47,5.48] + [-1.4,1.4]e-4 ε (54.07s/4Mb)
GC4	105	56 (53/3)	[-∞,∞] + [-∞,∞]ε (4.72s/10Mb)	[9.99,10.0] + [-3.2,3.1]e-5 ε (1.11s/7Mb)
GC10	105	189 (186/3)	[-∞,∞] + [-∞,∞]ε (22.18s/15Mb)	[54.97,55.03] + [-∞,∞]ε (15.6s/23Mb)
A2	576	75 (59/16)	[6.523,6.524] + [-5.5,5.6]e-6 ε (1.43s/9Mb)	[6.523,6.524] + [-5.5,5.6]e-6 ε (2.4s/13Mb)

```
double xi, xsi, A, temp;
signed int *PtrA, *Ptrxi, cond, exp, i;
A = __BUILTIN_DAED_DBETWEEN(20.0,30.0);
PtrA = (signed int *) (&A); Ptrxi = (signed int *) (&xi);
exp = (signed int) ((PtrA[0] & 0x7FF00000) >> 20) - 1023;
xi = 1; Ptrxi[0] = ((1023-exp) << 20);
cond = 1; i = 0;
while (cond) {
  xsi = 2*xi-A*xi*xi; temp = xsi-xi;
  cond = ((temp > e-10) || (temp < -e-10));
  xi = xsi; i++; }
```

Fig. 2. Newton method for computing $\frac{1}{A}$

`Poly` is the computation of a polynomial of degree 4, not in Horner form, from an initial interval. `Inv` is the program we depicted above. `F1a` and `F1b` are two linear recursive filters of order 1. `F1b` is almost ill-conditionned, and needs an enormous amount of virtual unrollings to converge in interval semantics (we use 5000 unfoldings of the main loop, in the line below the entry corresponding to `F1b`, named `idem`). The potentially infinite error found by our current implementation of affine forms, in `F1a` and `F1b`, is due to the fact we do not have a relational analysis on errors yet. `F2` is a linear recursive filter of order 2. `SA` and `SH` are two methods for computing the square root of a number, involving iterative computations of polynomials (in `SH`, of order 5). `GC4` and `GC10` are gradient conjugate algorithms (iterations on expressions involving division of multivariate polynomials of order 2), for a set of initial matrices "around" the discretisation of a 1-dimensional Laplacian, with a set of initial conditions, in dimensions 4x4 and 10x10 respectively in `GC4` and `GC10`. `A2` is a sample of an industrial program, involving filters, and mostly simple iterative linear computations.

5 Conclusion

In this paper, we introduced a new domain which gives tight enclosures for both floating-point and real value semantics of programs. This domain has been implemented in our static analyzer Fluctuat, which is used in an industrial context.

As we see from the examples of section 4, it always provides much more precise results than the interval based abstract domain of [4], at a small memory expense, and sometimes even faster. Notice that our domain is in no way specialized, and works also well on non-linear iterative schemes. As far as we know, no current static analyzer is able to find as tight enclosures for such computations as we do, not mentionning that we are also analyzing the difference between floating-point and real number semantics. The only comparable work we know of, for bounding the floating-point semantics, is the one of [1]. But the approach in [1] is more specialized, and would probably compare only on first and second order linear recursive filters.

Current work includes relational methods for the error computation, as quickly hinted in [13] (it should be noted that the computation of values will also benefit from the relational computation of errors), and better heuristics for join, meet and fixed point approximations in the domain of affine forms. We are also working on underapproximations relying on the same kind of domains.

References

1. B. Blanchet, P. and R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux and X. Rival. A static analyzer for large safety-critical software. PLDI 2003.
2. J. Stolfi and L. H. de Figueiredo. An introduction to affine arithmetic. TEMA Tend. Mat. Apl. Comput., 4, No. 3 (2003), pp 297-312.
3. P. Cousot and R. Cousot. Abstract interpretation frameworks. Journal of Logic and Symbolic Computation, 2(4), 1992, pp 511-547.
4. E. Goubault. Static analyses of the precision of floating-point operations. In Static Analysis Symposium, SAS'01, number 2126 in LNCS, Springer-Verlag, 2001.
5. E. Goubault, M. Martel, and S. Putot. Asserting the precision of floating-point computations : a simple abstract interpreter. In ESOP'02, LNCS, Springer 2002.
6. E. Goubault, M. Martel, and S. Putot. Some future challenges in the validation of control systems. In European Symposium on Real-Time Systems ERTS'06.
7. S. Sankaranarayanan, M. Colon, H. Sipma and Z. Manna. Efficient strongly relational polyhedral analysis. In Proceedings of VMCAI, to appear 2006.
8. M. Martel. Propagation of roundoff errors in finite precision computations : a semantics approach. In ESOP'02, number 2305 in LNCS, Springer-Verlag, 2002.
9. A. Miné. The octagon abstract domain. In Journal of Higher-Order and Symbolic Computation, to appear 2006.
10. A. Miné. Relational abstract domains for the detection of floating-point run-time errors. In ESOP'04, number 2986 in LNCS, Springer-Verlag, 2004.
11. MPFR library Documentation and downloadable library at http://www.mpfr.org.
12. S. Putot, E. Goubault and M. Martel. Static analysis-based validation of floating-point computations. In LNCS 2991, Springer-Verlag, 2004.
13. S. Putot, E. Goubault. Weakly relational domains for floating-point computation analysis. In Proceedings of NSAD, 2005.