Noname manuscript No.
(will be inserted by the editor)

Enhancing the Implementation of Mathematical Formulas for Fixed-Point and Floating-Point Arithmetics

Matthieu Martel

the date of receipt and acceptance should be inserted later

Abstract This article introduces some techniques to estimate and to improve the numerical quality of computations performed using different computer arithmetics. A general methodology is introduced and it is applied to the fixed-point and floating-point formats. We show how to globally measure the quality of the implementation of a formula with respect to some quality indicators. In the case of the floating-point arithmetic, the indicator measures the distance between the computer and exact results in the worst case. In the case of the fixed-point arithmetic, the indicator bounds the number of digits needed to represent all the intermediary results. Next, we show how the operations which make mostly decrease the quality of an indicator can be identified. This information helps the programmer to improve the implementation by underlying the main sources of degradation. Finally, we introduce a fully automatic expression transformation technique to rewrite a formula into a better, mathematically equivalent one. The new formula is more accurate than the original one with respect to the chosen quality indicator.

Keywords Numerical precision, Static Analysis, Abstract Interpretation, Program Transformation.

1 Introduction

In general, the computations carried out on machines are approximative because of the finite representation of numbers. Then an obvious question is how to estimate the quality of a certain implementation of a formula and how to enhance it. For example, one may wish to measure the absolute or relative precision of the result of a sequence of operations, assuming ranges for the inputs and assuming that the operations are performed in the floating-point arithmetic described by the IEEE754 Standard [1]. If a

Matthieu Martel
Laboratoire ELIAUS-DALI
Université de Perpignan Via Domitia
52 avenue Paul Alduy
66860 Perpignan Cedex, France
E-mail: matthieu.martel@univ-perp.fr

fixed-point arithmetic is used, one may rather aim at minimizing the number of digits required to reach a given precision. these problems have many industrial applications: floating-point numbers are more and more used in safety-critical software [6] and fixed-point numbers are widely used in many, critical or not, embedded systems (from cellular phones to vehicles).

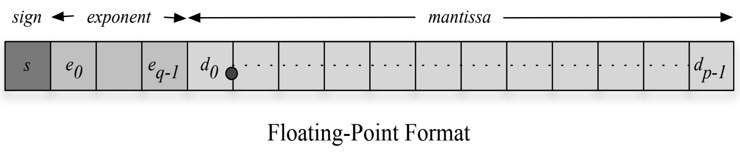
Understanding the reasons why the implementation of a formula is numerically bad and how to improve it is usually difficult because computer arithmetics are particularly not intuitive. For example, in floating-point arithmetic, the elementary operations (e.g. addition, multiplication…) are not associative, invertible, distributive, etc. [7,18]. So, it is necessary to provide tools to the programmers, in order to help them to increase the numerical quality of their codes. During the last few years, static analyses by abstract interpretation [3] of the numerical accuracy of floating-point computations have been introduced [8,14,16] and implemented in the Fluctuat tool [9,10] which is used in many industrial contexts. A main advantage of this method is that it enables one to bound safely all the errors arising during a computation, for large ranges of inputs. It also provides hints on the sources of errors, that is on the operations which introduce the most important precision loss. This latter information is of great interest to improve the accuracy of the implementation. Other methods, not based on static analysis, are compared in [15]. semantics-based program transformation [4,12] for floating-point arithmetic expressions has been introduced [17]. This method enables one to automatically rewrite a general formula into another mathematically equivalent and more precise formula. In this area, there only exists improvement methods dedicated to specific classes of formulas, for example to improve the evaluation of polynomial expressions [2,13].

Regardless of the arithmetic used, tools are necessary to help the programmers to determine which operations reduce the quality of an implementation and how to improve it. This is the main purpose of this article: we consider computer arithmetics extended by quality indicators. The largest the indicator is, the worst the implementation is. Then we show how to detect automatically which operations mainly contribute to make the indicators grow and how to rewrite an expression into a more efficient one with respect to the chosen indicator. Detecting the places where quality is lost is a generalisation of the work on error series introduced to assert the accuracy of floating-point expressions [16,9]. The enhancement of the implementation is a generalisation of the semantics-based transformation of [17]. We apply our framework to two different arithmetics: the floating-point arithmetic and the fixed-point arithmetic. In the former case, the indicator concerns the precision of the computation. In the latter case, the indicator concerns the number of digits required to perform an exact computation.

This article is organised as follows. Section 2 briefly introduces the floating-point and fixed-point arithmetics. In Section 3, we introduce an abstract semantics which computes the indicators used to estimate the quality of a formula. Section 4 introduces more subtle abstract semantics to detect the operations which lower the quality of an implementation. Finally, in Section 5, we introduce a semantics-based transformation which makes it possible to automatically rewrite formulas into more efficient ones. All these methods are applied to the floating-point and fixed-point arithmetics. Section 6 concludes.

2 Computer Arithmetics

This section briefly surveys the aspects of floating-point and fixed-point arithmetics useful to the comprehension of the rest of this article. It also defines the quality indicators that we aim at improving for each format.



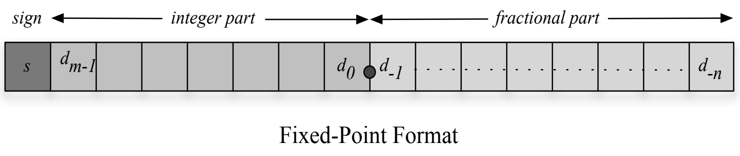


Fig. 1 Representation of the floating-point and fixed-point formats.

2.1 Floating-Point Arithmetic and the IEEE754 Standard

The IEEE754 Standard specifies the representation of numbers and the semantics of the elementary operations for floating-point arithmetic [1, 7]. It is implemented in most of modern general-purpose processors. First of all, a floating-point number x in base β is defined by

$$x = s \cdot (d_0.d_1 \dots d_{p-1}) \cdot \beta^e = s \cdot m \cdot \beta^{e-p+1} \tag{1}$$

where $s \in \{-1, 1\}$ is the sign, $m = d_0 d_1 \dots d_{p-1}$ is the mantissa with digits $0 \leq d_i < \beta$, $0 \leq i \leq p-1$, p is the precision and e is the exponent, $e_{min} \leq e \leq e_{max}$ (see Figure 1). A floating-point number x is normalized whenever $d_0 \neq 0$. Normalization avoids multiple representations of the same number. IEEE754 Standard introduces a few values for p, e_{min} and e_{max}. For example, simple precision numbers are defined by $\beta = 2$, $p = 23$, $e_{min} = -126$ and $e_{max} = +127$. The IEEE754 Standard also specifies special values (denormalized numbers, infinites and NaN) which are not used in this article.

Let $\uparrow_\circ : \mathbb{R} \to \mathbb{F}$ be the function which returns the roundoff of a real number following the rounding mode $\circ \in \{\circ_{-\infty}, \circ_{+\infty}, \circ_0, \circ_\sim\}$ (towards $\pm\infty$, 0 or to the nearest). $\uparrow_\circ$ is fully specified by the IEE754 Standard which also requires, for any elementary operation $\Diamond$, that:

$$x_1 \; \Diamond_{\mathbb{F},\circ} \; x_2 \; = \uparrow_\circ (x_1 \; \Diamond_{\mathbb{R}} \; x_2) \tag{2}$$

Equation (2) states that the result of an operation between floating-point numbers is the roundoff of the exact result of this operation. In this article, we also use the function $\downarrow_\circ : \mathbb{R} \to \mathbb{R}$ which returns the roundoff error. We have:

$$\downarrow_\circ (r) = r - \uparrow_\circ (r) \tag{3}$$

In floating-point arithmetic, enhancing the quality of the implementation of a formula $f(\mathbf{x})$ then consists of minimizing the roundoff error on the result. In other words, using the notation of Equation (3), we aim at minimizing $\downarrow_\circ (f(\mathbf{x}))$, for all the possible vectors of inputs $\mathbf{x}$.

2.2 Fixed-Point Arithmetic

There is no general standard for fixed-point arithmetic comparable to the IEEE754 Standard. Following [19,20], we consider that a number x is written:

$$x = s \cdot (d_{m-1} \dots d_0 . d_{-1} \dots d_{-n}) = s \cdot \sum_{i=-n}^{m-1} d_i \beta^i \tag{4}$$

In Equation (4), x is a number made of $m + n$ digits. The m first digits represent the integer part while the n last digits represent the fractional part (see Figure 1). $s \in \{-1, 1\}$ is the sign of x. For the basis, we always assume that $\beta = 2$. In addition, for the sake of simplicity and without lost of generality, we do not consider numbers encoded in the two's complement format which is also common in the implementations of fixed-point arithmetic. As argued in the next paragraph, we assume that no overflow arises and that, whenever it is necessary, the results of elementary operations are truncated (rounding mode towards zero).

Slightly different problems may be formulated concerning the enhancement of the implementation of a formula in fixed-point arithmetic. Following [19,20], in this article, we are interested in minimizing the m parameter of Equation (4), that is in finding the minimal size for the integer part of numbers such that no overflow arise during the computation, for all the acceptable inputs. So, we introduce the function

$$\leftarrow_\circ (x) = \min \left\{ m \in \mathbb{N} \ : \ \lfloor x \rfloor \leq \beta^m \right\} \tag{5}$$

where $\lfloor x \rfloor$ denotes the integer part of x. Implicitly, we consider that in the final implementation all the values are encoded in the same format (the one for which we determine m). This is usually the case when the program is targeted for a general purpose processor: the designer wants to know, for example, if 16 bits are enough to perform the computation or if all the data must be encoded in a 32 bits format. For more specific processing units (e.g. FPGAs), one may wish to minimize the size of the circuit by using numbers of different formats (e.g. to use less bits when the values are smaller). In this case, the way we compute the quality indicators for a fixed-point implementation should be slightly modified.

3 Measuring the Quality of an Implementation

In this section, we introduce two semantics for the arithmetic expressions whose syntax is given by:

$$e ::= v \mid e_1 + e_2 \mid e_1 - e_2 \mid e_1 \times e_2 \tag{6}$$

These semantics are related to our two quality indicators (see Section 2). In both cases, a value is a pair (x, μ) where x denotes the computer number, i.e. a fixed-point or a floating-point number, and μ measures the quality of x. In our case, μ will denote either the number of digits needed to encode the integer part of x in the fixed-point format (parameter m of Equation (4)) or the distance between a real number $x_{\mathbb{R}}$ and the floating-point number x corresponding to the roundoff of $x_{\mathbb{R}}$ (i.e. $\downarrow_\circ (x_{\mathbb{R}})$ as defined in Equation (3)). In addition, we consider a left-to-right evaluation order for the expression, as given by the straightforward reduction rules of Figure 2.

$$\frac{v = v_1 + v_2}{v_1 + v_2 \;\rightarrow\; v} \qquad \frac{e_1 \;\rightarrow\; e_1'}{e_1 + e_2 \;\rightarrow\; e_1' + e_2} \qquad \frac{e_2 \;\rightarrow\; e_2'}{v_1 + e_2 \;\rightarrow\; v_1 + e_2'}$$

$$\frac{v = v_1 \times v_2}{v_1 \times v_2 \;\rightarrow\; v} \qquad \frac{e_1 \;\rightarrow\; e_1'}{e_1 \times e_2 \;\rightarrow\; e_1' \times e_2} \qquad \frac{e_2 \;\rightarrow\; e_2'}{v_1 \times e_2 \;\rightarrow\; v_1 \times e_2'}$$

Fig. 2 Operational semantics of arithmetic expressions.

In the rest of this article, we consider abstract values $(x^\sharp, \mu^\sharp)$ where $x^\sharp$ and $\mu^\sharp$ are intervals. A value $(x^\sharp, \mu^\sharp)$ abstracts a set of concrete values $\{(x_i, \mu_i),\ i \in I\}$ by intervals in a component-wise way. The reduction rules of Figure 2 are left unchanged for the abstract semantics.

$$(x_1^\sharp, \mu_1^\sharp) + (x_2^\sharp, \mu_2^\sharp) = \Big(\uparrow^\sharp_\circ (x_1^\sharp + x_2^\sharp), \mu_1^\sharp + \mu_2^\sharp + \downarrow^\sharp_\circ (x_1^\sharp + x_2^\sharp)\Big) \tag{7}$$

$$(x_1^\sharp, \mu_1^\sharp) - (x_2^\sharp, \mu_2^\sharp) = \Big(\uparrow^\sharp_\circ (x_1^\sharp - x_2^\sharp), \mu_1^\sharp - \mu_2^\sharp + \downarrow^\sharp_\circ (x_1^\sharp - x_2^\sharp)\Big) \tag{8}$$

$$(x_1^\sharp, \mu_1^\sharp) \times (x_2^\sharp, \mu_2^\sharp) = \Big(\uparrow^\sharp_\circ (x_1^\sharp \times x_2^\sharp), x_1^\sharp \times \mu_2^\sharp + x_2^\sharp \times \mu_1^\sharp + \mu_1^\sharp \times \mu_2^\sharp + \downarrow^\sharp_\circ (x_1^\sharp \times x_2^\sharp)\Big) \tag{9}$$

Fig. 3 Abstract semantics of the elementary operations for the floating-point arithmetic.

The abstract semantics of arithmetic operations are given in Figure 3 and in Figure 4 for the floating-point and fixed-point arithmetics, respectively. In the floating-point case (Figure 3), we compute how the roundoff errors are propagated. $\downarrow^\sharp_\circ$ is a safe abstraction of $\downarrow_\circ$, i.e. $\forall x \in [\underline{x}, \overline{x}],\ \downarrow_\circ (x) \in \downarrow^\sharp_\circ ([\underline{x}, \overline{x}])$. For example, if the current rounding mode $\circ$ is to the nearest, one may choose

$$\downarrow^\sharp_\circ ([\underline{x}, \overline{x}]) = [-y, y] \quad \text{with } y = \frac{1}{2}\mathrm{ulp}\big(\max(|\underline{x}|, |\overline{x}|)\big) \tag{10}$$

where the *u*nit in the *l*ast *p*lace $\mathrm{ulp}(x)$ is the weight of the least significant digit of the floating-point number x [7]. For an addition, the errors on the operands are added to the error due to the roundoff of the result, as specified by Equation (2). For a subtraction, the errors on the operands are subtracted. Finally, the semantics of the multiplication comes from the development of $(x_1^\sharp + \mu_1^\sharp) \times (x_2^\sharp + \mu_2^\sharp)$.

In the fixed-point case (Figure 4), the measure $\mu^\sharp$ indicates the maximal number of bits needed to encode a value somewhere in the computation. In this case, we do not need an interval for $\mu^\sharp$ since we only store the greatest value. So, in the fixed-point abstract semantics $\mu^\sharp$ is an integer. To compute $\mu^\sharp$, we take the maximum of the measures $\mu_1^\sharp$ and $\mu_2^\sharp$ on the operands and of the measure $\leftarrow^\sharp_\circ (x^\sharp)$ on the result $x^\sharp$ of an operation, where $\leftarrow^\sharp_\circ (x)$ stands for a safe abstraction of $\leftarrow_\circ (x)$, i.e. $\forall x \in [\underline{x}, \overline{x}],\ \leftarrow_\circ (x) \in \leftarrow^\sharp_\circ ([\underline{x}, \overline{x}])$.

$$(x_1^\sharp, \mu_1^\sharp) + (x_2^\sharp, \mu_2^\sharp) = \left(x_1^\sharp + x_2^\sharp, \max(\mu_1^\sharp, \mu_2^\sharp, \hookleftarrow_\circ^\sharp (x_1^\sharp + x_2^\sharp))\right) \quad (11)$$

$$(x_1^\sharp, \mu_1^\sharp) - (x_2^\sharp, \mu_2^\sharp) = \left(x_1^\sharp - x_2^\sharp, \max(\mu_1^\sharp, \mu_2^\sharp, \hookleftarrow_\circ^\sharp (x_1^\sharp - x_2^\sharp))\right) \quad (12)$$

$$(x_1^\sharp, \mu_1^\sharp) \times (x_2^\sharp, \mu_2^\sharp) = \left(x_1^\sharp \times x_2^\sharp, \max(\mu_1^\sharp, \mu_2^\sharp, \hookleftarrow_\circ^\sharp (x_1^\sharp \times x_2^\sharp))\right) \quad (13)$$

Fig. 4 Abstract semantics of the elementary operations for the fixed-point arithmetic.

We end this section by considering the following example: Let

$$\mathtt{E} = (\mathtt{a} + (\mathtt{b} + (\mathtt{c} + \mathtt{d}))) \times \mathtt{e} \quad (14)$$

and let us assume that the variables belong to the ranges:

$$\begin{array}{lll} \mathtt{a} \in [-14, -13] & \mathtt{b} \in [-3, -2] & \\ \mathtt{c} \in [3, 3.5] & \mathtt{d} \in [12.5, 13.5] & \mathtt{e} = 2 \end{array} \quad (15)$$

Using the semantics of Figure 3 and Figure 4, we obtain the following results:

– Floating-point arithmetic:

$$E_{\text{float}} = \big([-3, 4], [-2.861022949 \cdot 10^{-6}, 0]\big)$$

This value indicates that the result returned by the machine always belongs to the interval $[-3, 4]$ and that, for any combination of inputs taken in the correct ranges, the roundoff error on the result is always less than $2.861022949 \cdot 10^{-6}$ in absolute value.

– Fixed-point arithmetic:

$$E_{\text{fixed}} = \big([-3, 4], 5\big)$$

Just like in the previous case, this value states that the result returned by the machine always belongs to the interval $[-3, 4]$. In addition, it states that 5 bits may be needed for the integer part, somewhere in the computation. Note that only 3 bits are required for the integer part of the result. However, for instance, if $\mathtt{c} = 3.5$ and $\mathtt{d} = 13.5$ then $\mathtt{c}+\mathtt{d} = 17$ and $\hookleftarrow_\circ (17) = 5$.

Given a value $(x^\sharp, \mu^\sharp)$, the indicator $\mu^\sharp$ measures the quality of the implementation of a formula in floating-point or fixed-point arithmetics, assuming that the inputs belong to certain ranges. In the next sections we introduce some techniques enabling the programmer to improve these indicators.

4 Semi-Automatic Improvement of the Quality

In this section, we introduce new semantics to trace the operations of a formula which mostly lower the quality of an implementation. This generalizes the semantics of Section 3 which computes the indicator μ but do not indicate how it is obtained. The semantics introduced here are said *semi-automatic* since they detect the places where the quality is lost. This is an important help for the programmer who aims at improving a code. However this is not a fully automatic method in the sense that no solution is given concerning how to enhance the quality of the implementation.

The semantics introduced here were first introduced for the floating-point arithmetic under the name of error series semantics [8,14]. First of all, in order to trace the sources of quality loss, labels are attached to the terms. A labeled expression e^{ℓ} is then defined by:

$$e^{\ell} ::= v^{\ell} \mid e_1^{\ell_1} +^{\ell} e_2^{\ell_2} \mid e_1^{\ell_1} -^{\ell} e_2^{\ell_2} \mid e_1^{\ell_1} \times^{\ell} e_2^{\ell_2} \tag{16}$$

We introduce now the values of the semantics of indicator series. A value $x_{\mathbb{S}}$ is a pair

$$x_{\mathbb{S}} = \big(x, \langle \mu_{\ell_1}, \ldots, \mu_{\ell_n}, \mu_{\hbar} \rangle\big) \tag{17}$$

where x denotes the computer representation of the number $x_{\mathbb{S}}$ and the tuple $\langle \mu_{\ell_1}, \ldots,$ $\mu_{\ell_n}, \mu_{\hbar} \rangle$ has one component per label $\ell_1, \ldots, \ell_n$ used in the expressions plus one component reserved for a special label $\hbar$ used for the higher-order terms introduced by non-linear computations.

$$\begin{gathered} x_{\mathbb{S}}^{\sharp} +^{\ell_k} y_{\mathbb{S}}^{\sharp} \\ = \\ \Big(\uparrow_\circ (x^\sharp + y^\sharp), \Big\langle \mu_{\ell_1}^{\sharp} + \nu_{\ell_1}^{\sharp}, \ldots, \mu_{\ell_k}^{\sharp} + \nu_{\ell_k}^{\sharp} + \downarrow_\circ^{\sharp} (x^\sharp + y^\sharp), \ldots, \mu_{\ell_n}^{\sharp} + \nu_{\ell_n}^{\sharp}, \mu_{\hbar}^{\sharp} + \nu_{\hbar}^{\sharp} \Big\rangle\Big) \end{gathered} \tag{18}$$

$$\begin{gathered} x_{\mathbb{S}}^{\sharp} -^{\ell_k} y_{\mathbb{S}}^{\sharp} \\ = \\ \Big(\uparrow_\circ (x^\sharp - y^\sharp), \Big\langle \mu_{\ell_1}^{\sharp} - \nu_{\ell_1}^{\sharp}, \ldots, \mu_{\ell_k}^{\sharp} - \nu_{\ell_k}^{\sharp} + \downarrow_\circ \sharp(x^\sharp - y^\sharp), \ldots, \mu_{\ell_n}^{\sharp} - \nu_{\ell_n}^{\sharp}, \mu_{\hbar}^{\sharp} - \nu_{\hbar}^{\sharp} \Big\rangle\Big) \end{gathered} \tag{19}$$

$$\begin{aligned} x_{\mathbb{S}}^{\sharp} \times^{\ell_k} y_{\mathbb{S}}^{\sharp} = (\, \uparrow_\circ (x^\sharp \times y^\sharp),\ \langle & x^\sharp \nu_{\ell_1}^{\sharp} + y^\sharp \mu_{\ell_1}^{\sharp}, \ldots, \\ & x^\sharp \nu_{\ell_k}^{\sharp} + y^\sharp \mu_{\ell_k}^{\sharp} + \downarrow_\circ^{\sharp} (x^\sharp \times y^\sharp), \ldots, \\ & x^\sharp \nu_{\ell_n}^{\sharp} + y^\sharp \mu_{\ell_n}^{\sharp}, \mu_{\hbar}^{\sharp} \nu_{\hbar}^{\sharp} + \textstyle\sum_{i=1}^{n} \Big(\sum_{j=1}^{n} \mu_{\ell_i}^{\sharp} \nu_{\ell_j}^{\sharp}\Big) \rangle) \end{aligned} \tag{20}$$

Fig. 5 Abstract semantics of elementary operations for the indicator series in the floating-point arithmetic.

Intuitively, in the case of the floating-point arithmetic, in the tuple of Equation (17), the term μ_{ℓ_k} represents the contribution to the global error computed by the semantics of Section 3 of the error introduced by the control point ℓ_k. So, in the concrete semantics, we may relate the measure μ to Equation (17) by the property:

$$\mu = \mu_{\hbar} + \sum_{i=1}^{n} \mu_{\ell_i} \tag{21}$$

In other words, the error μ has been decomposed in a series or error terms which indicate to the programmer which elementary error has been mostly propagated in the computation and contributes mostly to the global error. The semantics of elementary operations is given Figure 5 for the floating-point arithmetic. The operands

$$x_{\mathbb{S}}^{\sharp} = \langle x^\sharp, \mu_{\ell_1}^{\sharp}, \ldots, \mu_{\ell_n}^{\sharp}, \mu_{\hbar}^{\sharp} \rangle$$

and

$$y_{\mathbb{S}}^{\sharp} = \langle y^\sharp, \nu_{\ell_1}^{\sharp}, \ldots, \nu_{\ell_n}^{\sharp}, \nu_{\hbar}^{\sharp} \rangle$$

are tuples of intervals representing abstract values. For an addition carried out at Label ℓ_k, the computer representable result $\uparrow_\circ^{\sharp} (x^\sharp + y^\sharp)$ is calculated and the indicator of each label is updated as follows: the new error related to a label $\ell \neq \ell_k$ is the sum of the errors related to ℓ in the operands, i.e. $\mu^\sharp_\ell + \nu^\sharp_\ell$. Next, the new error attached to ℓ_k is the sum of the errors on the operands plus the new error due to the addition labeled ℓ_k itself. The subtraction is very similar to the addition. For a product labeled ℓ_k, for each label $\ell \neq \ell_k$ we compute the propagation of the errors due to ℓ in the operands which yields $x^\sharp \nu^\sharp_\ell + y^\sharp \mu^\sharp_\ell$. For Label ℓ_k, the new error $\downarrow^\sharp_\circ (x^\sharp \times y^\sharp)$ introduced by the product is added to the former term.

$$x^\sharp_{\mathbb{S}} +^{\ell_k} y^\sharp_{\mathbb{S}} = \big(x^\sharp + y^\sharp, \langle\ \max(\mu^\sharp_{\ell_1}, \nu^\sharp_{\ell_1}), \ldots, \max\big(\mu^\sharp_{\ell_k}, \nu^\sharp_{\ell_k}, \leftarrow^\sharp_\circ (x^\sharp + y^\sharp)\big), \ldots, \max(\mu^\sharp_{\ell_n}, \nu^\sharp_{\ell_n})\rangle\big) \tag{22}$$

$$x^\sharp_{\mathbb{S}} -^{\ell_k} y^\sharp_{\mathbb{S}} = \big(x^\sharp - y^\sharp, \langle\ \max(\mu^\sharp_{\ell_1}, \nu^\sharp_{\ell_1}), \ldots, \max\big(\mu^\sharp_{\ell_k}, \nu^\sharp_{\ell_k}, \leftarrow^\sharp_\circ (x^\sharp - y^\sharp)\big), \ldots, \max(\mu^\sharp_{\ell_n}, \nu^\sharp_{\ell_n})\rangle\big) \tag{23}$$

$$x^\sharp_{\mathbb{S}} \times^{\ell_k} y^\sharp_{\mathbb{S}} = \big(x^\sharp \times y^\sharp, \langle\ \max(\mu^\sharp_{\ell_1}, \nu^\sharp_{\ell_1}), \ldots, \max\big(\mu^\sharp_{\ell_k}, \nu^\sharp_{\ell_k}, \leftarrow^\sharp_\circ (x^\sharp \times y^\sharp)\big), \ldots, \max(\mu^\sharp_{\ell_n}, \nu^\sharp_{\ell_n})\rangle\big) \tag{24}$$

Fig. 6 Abstract semantics of the elementary operations for the indicator series in the fixed-point arithmetic.

For the fixed-point arithmetic, the terms $\mu_{\ell_1} \ldots \mu_{\ell_n}$ of the tuple of Equation (17) are integers which indicate how many bits are needed to compute the integer part of the sub-expression of root ℓ_i for any i, $1 \le i \le n$. Here, the higher-order term $\mu_\hbar$ is always zero and we omit it in the formulas. The information collected by this semantics directly shows to the programmer which parts of a formula make the integer part of the fixed-point format grow. Again, this is useful to modify the implementation of the formula but this technique do not indicate how to achieve the enhancement.

The abstract semantics for the indicator series in fixed-point arithmetic is given in Figure 6. Basically, for an operation labeled ℓ_k, the size of the integer part is the maximum of the size of the operands and of the result. The other terms of the tuple are updated by keeping the maximum of the operands. As a result, the final tuple indicates the size of the integer parts of all the sub-expressions.

For example, we consider that the following labels are attached to the operations of the expression of Equation (14):

$$E^\ell = (\mathtt{a} +^{\ell_2} (\mathtt{b} +^{\ell_1} (\mathtt{c} +^{\ell_0} \mathtt{d}))) \times^{\ell_3} \mathtt{e}$$

Since we assume that there is no initial error on the data, the labels on the values `a`, `b`, `c`, `d` and `e` are useless. The abstract indicator series semantics gives the following results.

– Floating-point arithmetic:

$$\begin{aligned} E^\ell_{\text{float}} = \big([-3, 4], \langle\ & [-1.90734863281250 \cdot 10^{-6}, 1.90734863281250 \cdot 10^{-6}], \\ & [-9.53674316406250 \cdot 10^{-7}, 9.53674316406250 \cdot 10^{-7}], \\ & [-2.38418579101562 \cdot 10^{-7}, 2.38418579101563 \cdot 10^{-7}], \\ & [-2.38418579101562 \cdot 10^{-7}, 2.38418579101563 \cdot 10^{-7}]\rangle\big) \end{aligned} \tag{25}$$

The errors can be represented by an histogram, as shown in Figure 7. The labels $\ell_0 \ldots \ell_n$ are displayed on the x-axis and the measures are given by the y-axis. We

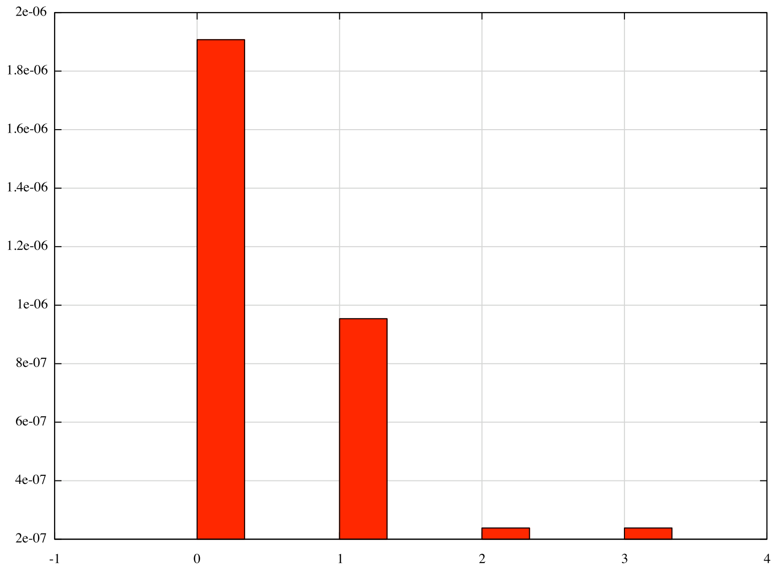


Fig. 7 Histogram for the indicator series of Equation (25): floating-point arithmetic.

can observe that the main errors are due to the first two additions labeled ℓ_0 and ℓ_1. In the floating-point format, `c` and `d` do not have the same exponent (in base 2 and their addition introduces an important roundoff error. The same phenomena arises at Point ℓ_1 but for smaller values. Clearly, in order to enhance the accuracy of this computation, the programmer should avoid to add `c` and `d`. In Section 5, we will introduce a way to transform this expression into a more precise one.

– Fixed arithmetic:

$$E^{\ell}_{\text{fixed}} = \big([-3,4], \langle 5,4,2,3\rangle\big) \tag{26}$$

$$E'^{\ell}_{\text{fixed}} = \big([-3,4], \langle 5,5,5,5\rangle\big) \tag{27}$$

Here, two series are relevant. They are drawn in the histogram of Figure 8. The series of Equation (27) is based on the semantics of Figure 6. In Equation (26), the terms $\max\big(\mu^{\sharp}_{\ell_k}, \nu^{\sharp}_{\ell_k}, \leftarrow^{\sharp}_{\circ}(x^{\sharp} \times y^{\sharp})\big)$ of equations (22) to (24) are replaced by $\leftarrow^{\sharp}_{\circ}(x^{\sharp} \times y^{\sharp})$. The observation of these series reveals that the computation must be carried out using 5 bits because of the result of the addition labeled ℓ_0. The other operations would require less bits. A better implementation of this formula in fixed-point arithmetic is given in Section 5.

5 Fully Automatic Improvement of the Quality

As discussed in Section 4, the indicator series provide information on where the quality is lost but no way to enhance the implementation is given. In this section, we introduce a program transformation which rewrites programs into more accurate ones with respect to a given quality indicator. In practice, we are going to apply this technique to the improvement of the precision of floating-point expressions and to the reduction of the size of the integer part of fixed-point numbers.

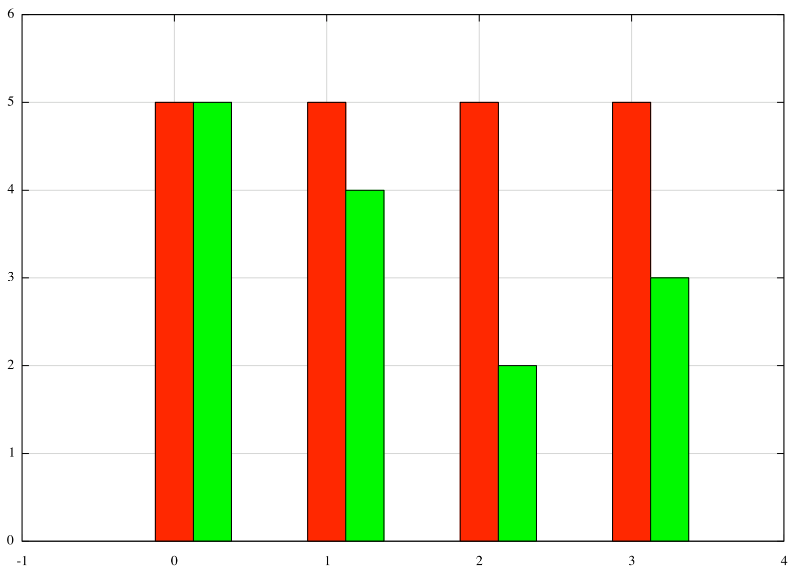


Fig. 8 Histogram for the indicator series of Equation (26) and Equation (27): fixed-point arithmetic.

(i)	$(e_1 + e_2) + e_3 \equiv e_1 + (e_2 + e_3)$
(ii)	$e_1 + e_2 \equiv e_2 + e_1$
(iii)	$e \equiv e + 0$
(iv)	$(e_1 \times e_2) \times e_3 \equiv e_1 \times (e_2 \times e_3)$
(v)	$e_1 \times e_2 \equiv e_2 \times e_1$
(vi)	$e \equiv e \times 1$
(vii)	$e_1 \times (e_2 + e_3) \equiv e_1 \times e_2 + e_1 \times e_3$

Fig. 9 Example of equivalence relation which may be used for the transformation of mathematic expressions.

Basically, the program transformation works as follows:

(i) The operational semantics of Figure 3 is extended by the rule:

$$\frac{e \equiv e_1 \quad e_1 \to e_2 \quad e_2 \equiv e'}{e \to e'} \tag{28}$$

where $\equiv$ is an relation which identifies mathematically equivalent expressions. For example, $\equiv$ may identify expressions which are equal up to associativity, symmetry and distributivity of the elementary operations (see Figure 9). Since there are usually many expressions e' equivalent to an expression e, the rule of Equation (28) makes the operational semantics non-deterministic, in the sense that from an expression e, many steps $e \to e'$ are possible, for syntactically different e'.

(ii) To limit the combinatorial explosion of the number of traces due to the rule of Equation (28), we introduce the set $\text{Expr}^\sharp_k$ of abstract expressions of height at most k and the abstraction function $\ulcorner . \urcorner^k : \text{Expr} \to \text{Expr}^\sharp$. From a formal point of view, $\text{Expr}^\sharp$ and $\ulcorner . \urcorner^k$ are defined in Figure 10. $\top_\eta$ denotes any expression. Note that, in abstract expressions, labels are attached to values (and only values). In

the abstract semantics, a new label, generated dynamically, is attached to the new values coming from the result of intermediary computations.

(iii) We extend the operational semantics by adding two environments to the expressions: a first environment $\rho^\sharp$: Lab $\to \wp(\text{Expr}^\sharp)$ maps the labels attached to values to abstract expressions. This indicates how the value has been calculated. The second environment $\sigma^\sharp$: Expr$^\sharp \to \mathcal{V}^\sharp$ maps abstract expressions to abstract values with global quality indicators, as defined in Section 3. The set $\mathcal{V}^\sharp$ denotes either the abstract domain of floating-point numbers or fixed-point numbers with global quality indicator. $\sigma(\eta)$ indicates the range of values in which are evaluated the expressions abstracted by η and encountered during the execution.

(iv) Finally, the program transformation consists of computing fully the abstract semantics. This semantics is non-deterministic because of (i) but the abstract expressions discussed in (ii) make the number of reductions polynomial. Using the information collected by the environments described in (iii), the result of each trace has a quality indicator. From the best quality indicator we can build a new expression, mathematically equivalent to the original one, by following actions attached to the reductions. These actions indicates which operation has actually been performed at each reduction step.

$$\begin{array}{l}
\eta_0 ::= v^{\sharp\ell} \mid \top_\eta \\
\eta_k ::= \eta_{k-1} \mid \eta_{k-1} + \eta_{k-1} \mid \eta_{k-1} \times \eta_{k-1}
\end{array}$$

$$\begin{array}{rcll}
\ulcorner v^\ell \urcorner^k &=& v^\ell & k \ge 0 \\
\ulcorner \top_\eta \urcorner^k &=& \top_\eta & k \ge 0 \\
\ulcorner e_1 + e_2 \urcorner^0 &=& \top_\eta & \\
\ulcorner e_1 \times e_2 \urcorner^0 &=& \top_\eta & \\
\ulcorner e_1 + e_2 \urcorner^k &=& \ulcorner e_1 \urcorner^{k-1} + \ulcorner e_2 \urcorner^{k-1} & k \ge 1 \\
\ulcorner e_1 \times e_2 \urcorner^k &=& \ulcorner e_1 \urcorner^{k-1} \times \ulcorner e_2 \urcorner^{k-1} & k \ge 1
\end{array}$$

Fig. 10 Abstract expressions and the abstraction function.

The abstract semantics resulting from the ideas detailed in the enumeration above is given in Figure 11. Its correctness is given, in the case of the floating-point arithmetic, in [17]. In Figure 11, $\lozenge$ denotes one of the elementary operations $+$, $-$ or $\times$ and $\sigma^\sharp \wedge [\eta \mapsto \nu]$ denotes the environment $\sigma^\sharp$ modified by $\sigma^\sharp(\eta) = \nu$. In a transition

$$(\rho^\sharp, \sigma^\sharp, e) \xrightarrow{A}_k (\rho'^\sharp, \sigma'^\sharp, e')$$

k denotes the user-defined parameter corresponding to the level where abstract expressions are cut and A is an action indicating which arithmetic operation has been performed at this step. Actions are used to rebuild a new expression from a trace. The relation $\equiv_k$ is the quotient $\equiv / \sim_k$ where $\sim_k$ is defined by

$$e \sim_k e' \iff \ulcorner e \urcorner^k = \ulcorner e' \urcorner^k.$$

The transformation τ_k is based on the minimal abstract trace $e \xrightarrow{A}{}^*_k v^\sharp$, i.e. the trace which yields the minimal abstract indicator $v^\sharp$. Because the semantics $\xrightarrow{A}_k$ allows more steps than the concrete semantics $\to$ (in $\to$ an expression may not be transformed

$$\frac{v^\sharp = \bigcup_{\substack{\eta_1 \in \rho^\sharp(\ell_1)\\ \eta_2 \in \rho^\sharp(\ell_2)}} \left(\sigma^\sharp(\eta_1)\Diamond^\sharp\sigma^\sharp(\eta_2)\right) \qquad E = \bigcup_{\substack{\eta_1 \in \rho^\sharp(\ell_1)\\ \eta_2 \in \rho^\sharp(\ell_2)}} \ulcorner\eta_1\Diamond\eta_2\urcorner^k \qquad \sigma^{\sharp\prime} = \sigma^\sharp \bigwedge_{\substack{\eta_1 \in \rho^\sharp(\ell_1),\ \eta_2 \in \rho^\sharp(\ell_2)\\ \eta = \ulcorner\eta_1\Diamond\eta_2\urcorner^k\\ \nu = \sigma^\sharp(\eta_1)\Diamond\sigma^\sharp(\eta_2)}} [\eta \mapsto \sigma^\sharp(\eta)\cup\nu]}{\langle\rho^\sharp,\sigma^\sharp, v_0^{\ell_0}\Diamond v_1^{\ell_1}\rangle \xrightarrow{\ell=\ell_1\Diamond\ell_2}_k \langle\rho^\sharp[\ell\mapsto\rho^\sharp(\ell)\cup E],\sigma^{\sharp\prime}, v^\ell\rangle} \tag{29}$$

$$\frac{\langle\rho^\sharp,\sigma^\sharp,e_0\rangle \xrightarrow{A}_k \langle\rho^{\sharp\prime},\sigma^{\sharp\prime},e_2\rangle}{\langle\rho^\sharp,\sigma^\sharp,e_0\Diamond e_1\rangle \xrightarrow{A}_k \langle\rho^{\sharp\prime},\sigma^{\sharp\prime},e_2\Diamond e_1\rangle} \tag{30}$$

$$\frac{e \equiv_k e_1 \qquad \langle\rho^\sharp,\sigma^\sharp,e_1\rangle \xrightarrow{A}_k \langle\rho^{\sharp\prime},\sigma^{\sharp\prime},e_2\rangle \qquad e_2 \equiv_k e_3}{\langle\rho^\sharp,\sigma^\sharp,e_0\rangle \xrightarrow{A}_k \langle\rho^{\sharp\prime},\sigma^{\sharp\prime},e_3\rangle} \tag{31}$$

Fig. 11 The abstract semantics.

by $\equiv_k$), we cannot directly transform a trace of $\xrightarrow{A}_k$ into a trace of $\rightarrow$: we first have to rebuild the totally parsed expression which has actually been evaluated by $\xrightarrow{A}_k$. This is achieved by using the actions A appearing in the transitions of the abstract semantics and which collect the operations actually performed along the traces.

Actions are expressions of the form $\ell = \ell_1 + \ell_2$ or $\ell = \ell_1 \times \ell_2$, where ℓ, ℓ_1 and ℓ_2 are labels belonging to Lab and attached to values. For example, an action $\ell = \ell_1 + \ell_2$ indicates that the value of label ℓ is the addition of the expressions of labels ℓ_1 and ℓ_2.

$$\mathsf{P}\left(\langle\rho^\sharp,\sigma^\sharp,e\rangle \xrightarrow{\ell=\ell_1+\ell_2}_k \langle\rho^{\sharp\prime},\sigma^{\sharp\prime},e'\rangle, \iota\right) = \iota[\ell\mapsto\iota(\ell_1)+\iota(\ell_2)] \tag{32}$$

$$\mathsf{P}\left(\langle\rho^\sharp,\sigma^\sharp,e\rangle \xrightarrow{\ell=\ell_1\Diamond\ell_2}_k \langle\rho^{\sharp\prime},\sigma^{\sharp\prime},e'\rangle, \iota\right) = \iota[\ell\mapsto\ell_1\times\ell_2] \tag{33}$$

$$\mathsf{P}\left(\langle\rho^\sharp,\sigma^\sharp,v^{\sharp\ell}\rangle, \iota\right) = \iota(\ell) \tag{34}$$

$$\mathsf{P}\left(s_1 \xrightarrow{A}_k s_2 \xrightarrow{A}_k \ldots s_n, \iota\right) = \mathsf{P}\left(s_2 \xrightarrow{A}_k \ldots s_n, \mathsf{P}(s_1 \xrightarrow{A}_k s_2)\right) \tag{35}$$

Fig. 12 Generation of the new expression.

The expression generation function P is defined in Figure 12. P takes a trace, an environment ι : Lab $\rightarrow$Expr and computes a new environment ι'. For a trace $t^\sharp = \langle\rho^\sharp, s^\sharp, e\rangle \xrightarrow{A}{}_k^* \langle\rho^{\sharp\prime},\sigma^{\sharp\prime},v^\sharp\rangle$, initially assuming that $\iota(\ell) = v$ for any value v^ℓ occurring in the source expression e, $\mathsf{P}(t^\sharp,\iota) = \iota'(\ell)$, where $\iota'(\ell)$ is the expression actually evaluated by $t^\sharp$.

Applied to the expression of Equation (14), our transformation technique yields the following results:

– Floating-point arithmetic: $\mathtt{E}$ is transformed into the new expression

$$\mathtt{E'}_{\text{float}} = ((\mathtt{a}+\mathtt{b})\times\mathtt{e}) + ((\mathtt{c}+\mathtt{d})\times\mathtt{e}) \tag{36}$$

and the global quality indicator attached to $\mathtt{E}_{\text{float}}$ is:

$$E_{\text{float}} = \left([-3,4],[-2.384185791\cdot10^{-6},-1.430511475\cdot10^{-6}]\right) \tag{37}$$

Remark that, in $\mathtt{E}_{\text{float}}$, the variables a and b (resp. c and d) added first which does not correspond to the original parsing. However, since $(\mathtt{a}+\mathtt{b}) \approx (\mathtt{c}+\mathtt{d})$ this writing reduces the roundoff errors. In addition the multiplication has been distributed. This avoids to multiply some of the roundoff errors due to the additions of the original formula.

– Fixed-point arithmetic:

$$\mathtt{E'}_{\text{fixed}} = \mathtt{e} \times ((\mathtt{a}+\mathtt{d}) + (\mathtt{b}+\mathtt{c})) \tag{38}$$

$$E'_{\text{fixed}} = \big([-3,4],4\big) \tag{39}$$

In $\mathtt{E'}$, a and d are added first and the product is not distributed. These choices make it possible to store all the intermediate values on 4 bits only (in absolute value, the greatest number arising during the computation is 14).

Note that, while $\mathtt{E'_{\text{float}}}$ and $\mathtt{E'_{\text{fixed}}}$ are quite different, they have been obtained automatically, from the same algorithm, based on the relation $\equiv_k$. The only difference in the computation concerns the indicator used to estimate the quality of the formulas. Both transformations enhance the quality of the implementation with respect to the chosen indicator.

6 Conclusion

In this article, we have extended recent work on the enhancement of the implementation of floating-point expressions to the case of fixed-point arithmetic. This is done in a general framework, where quality indicators are attached to the values manipulated by the computer. Swapping from one arithmetic to another only involves to change the quality indicator, the semantics being left unchanged. Our running example enables to compare the results of all the analyses and transformations. Our approach is not specific to the arithmetics used in this article and can be used in other cases. For example, we could also consider the case of interval arithmetic [11]: intervals often yield pessimistic results because of the wrapping effect and because of the lack of information on the relations between variables. For example, a direct evaluation of the polynomial

$$P(x) = x - x^2$$

with $x \in [0,1]$ yields $P(x) = [-1,1]$. Using Horner scheme, we obtain $P(x) = x(1-x) = [0,1]$. Finally, the most precise solution is $P(x) = [0, \frac{1}{4}]$. In our framework, we can define a new quality indicator corresponding to the width of the interval: in this case, a value is a pair $([a,b], |b-a|)$ where $[a,b]$ if an interval value and $|b-a|$ is the indicator. For example, using the values of Equation (15) for a, b, c, d and setting e$= [-2,2]$, we obtain, for the different formulas encountered in this article:

– $\mathtt{E} = (\mathtt{a} + (\mathtt{b} + (\mathtt{c}+\mathtt{d}))) \times \mathtt{e}$: $([-4,4],8)$,
– $\mathtt{E'}_{\text{float}} = ((\mathtt{a}+\mathtt{b}) \times \mathtt{e}) + ((\mathtt{c}+\mathtt{d}) \times \mathtt{e})$: $([-64,68],132)$,
– $\mathtt{E'}_{\text{fixed}} = \mathtt{e} \times ((\mathtt{a}+\mathtt{d}) + (\mathtt{b}+\mathtt{c}))$: $([-4,4],8)$.

The quality indicator of the first and third formulas is 8. Obviously, it is better than the indicator of the second formula which is 132. So, for the interval arithmetic, the programmer should not implement the formula by $\mathtt{E'}_{\text{fixed}}$.

More generally, we believe that yet other quality indicators could be interesting to study, for other kinds of fixed-point arithmetics but also, for example, for code obfuscation [5], execution-time, memory-consumption, etc. If many indicators are relevant for the same implementation, then multi-criteria enhancement techniques should also be explored.

Another research direction concerns the transformation of full programs: the indicator series based abstract semantics is used in industrial contexts, to validate safety-critical software. In the future, we aim at implementing a program transformer able to handle large codes. This tool should work both in floating-point and fixed-point arithmetics since there are many industrial needs in both contexts.

References

1. ANSI/IEEE. *IEEE Standard for Binary Floating-point Arithmetic*, std 754-1985 edition, 1985.
2. M. Ceberio and V. Kreinovich. Greedy algorithms for optimizing multivariate horner schemes. *ACM-SIGSAM Bulletin*, 38(1):8–15, 2004.
3. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximations of fixed points. In *Principles of Programming Languages 4*, pages 238–252. ACM Press, 1977.
4. P. Cousot and R. Cousot. Systematic design of program transformation frameworks by abstract interpretation. In *Conference Record of the Twentyninth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 178–190, Portland, Oregon, 2002. ACM Press, New York, NY.
5. M. Dalla Preda and R. Giacobazzi. Control code obfuscation by abstract interpretation. In *International Conference on Software Engineering and Formal Methods, SEFM'05*, pages 301–310. IEEE Computer Society Press, 2005.
6. D. Delmas and J. Souyris. Astré: From research to industry. In *Static Analysis Symposium, SAS'07*, number 4634 in Lecture Notes in Computer Science. Springer-Verlag, 2007.
7. D. Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 23(1):5–48, 1991.
8. E. Goubault. Static analyses of the precision of floating-point operations. In *Static Analysis Symposium, SAS'01*, number 2126 in Lecture Notes in Computer Science, pages 234–259. Springer-Verlag, 2001.
9. E. Goubault, M. Martel, and S. Putot. Asserting the precision of floating-point computations: a simple abstract interpreter. In 11^{th} *European Symposium on Programming, ESOP'02*, number 2305 in Lecture Notes in Computer Science, pages 209–212, 2002.
10. E. Goubault, M. Martel, and S. Putot. Some future challenges in the validation of control systems. In *Proceedings of the European Congress on Embedded Real Time Software (ERTS'06)*, 2006.
11. L. Jaulin, M. Kieffer, O. Didrit, and E. Walter. *Applied Interval Arithmetics with Examples in Parameter and State Estimation, Robust Control and Robotics.* Springer-Verlag, 2001.
12. N. Jones, C. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice Hall International, Int. Series in Computer Science, 1993.
13. P. Langlois and N. Louvet. How to ensure a faithful polynomial evaluation with the compensated Horner algorithm? In P. Kornerup and J.-M. Muller, editors, *18th IEEE International Symposium on Computer Arithmetic*, pages 141–149. IEEE Computer Society, June 2007.
14. M. Martel. Propagation of roundoff errors in finite precision computations: a semantics approach. In 11^{th} *European Symposium on Programming, ESOP'02*, number 2305 in Lecture Notes in Computer Science, pages 194–208. Springer-Verlag, 2002.
15. M. Martel. An overview of semantics for the validation of numerical programs. In *Verification, Model Checking and Abstract Interpretation, VMCAI'05*, number 3385 in Lecture Notes in Computer Science, pages 59–77. Springer-Verlag, 2005.
16. M. Martel. Semantics of roundoff error propagation in finite precision calculations. *Journal of Higher Order and Symbolic Computation*, 19:7–30, 2006.

17. M. Martel. Semantics-based transformation of arithmetic expressions. In *Static Analysis Symposium, SAS'07*, number 4634 in Lecture Notes in Computer Science. Springer-Verlag, 2007.
18. D. Monniaux. The pitfalls of verifying floating-point computations. *TOPLAS*, 30(3), May 2008.
19. R. Rocher, D. Menard, N. Herve, and Sentieys. Fixed-point configurable hardware components. *EURASIP Journal on Embedded Systems (JES)*, 2006, 2006.
20. R. Rocher, D. Menard, O. Sentieys, and P. Scalart. Analytical accuracy evaluation of fixed-point systems. In *EUSIPCO'07Poznan, Pologne*, 2007.