A New Abstract Domain for the Representation of Mathematically Equivalent Expressions†

Arnault Ioualalen[1,2,3] and Matthieu Martel[1,2,3]

[1] Univ. Perpignan Via Domitia, Digits, Architectures et Logiciels Informatiques, F-66860, Perpignan, France
[2] Univ. Montpellier II, Laboratoire d'Informatique Robotique et de Microélectronique de Montpellier, UMR 5506, F-34095, Montpellier, France
[3] CNRS, Laboratoire d'Informatique Robotique et de Microélectronique de Montpellier, UMR 5506, F-34095, Montpellier, France
arnault.ioualalen@univ-perp.fr, matthieu.martel@univ-perp.fr

Abstract. Exact computations being in general not tractable for computers, they are approximated by floating-point computations. This is the source of many errors in numerical programs. Because the floating-point arithmetic is not intuitive, these errors are very difficult to detect and to correct by hand and we consider the problem of automatically synthesizing accurate formulas. We consider that a program would return an exact result if the computations were carried out using real numbers. In practice, roundoff errors arise during the execution and these errors are closely related to the way formulas are written. Our approach is based on abstract interpretation. We introduce Abstract Program Equivalence Graphs (APEGs) to represent in polynomial size an exponential number of mathematically equivalent expressions. The concretization of an APEG yields expressions of very different shapes and accuracies. Then, we extract optimized expressions from APEGs by searching the most accurate concrete expressions among the set of represented expressions.

1 Introduction

In computers, exact computations are approximated by the floating-point arithmetic which relies on a finite representation of the numbers [1, 12, 15]. Although this approximation is often accurate enough, in some cases, it may lead to irrelevant or too inaccurate results. In programs, these roundoff errors are very difficult to understand and to rectify by hand. At least this task is strongly time consuming and, sometimes, it is almost impossible. Recently, validation techniques based on abstract interpretation [2] have been developed to assert the numerical accuracy of floating-point computations and to help the programmer

† This work was partly supported by the SARDANES project from the french Aeronautic and Space National Foundation.

to correct their codes [11, 10]. For example, Fluctuat is a static analyzer that computes the inaccuracies of floating-point computations in C codes and helps to understand their origin [5, 6]. This tool has been successfully used in many industrial projects, in aeronautics and other industries [4]. However, this method does not indicate how to correct programs in order to produce smaller errors. It is up to the programmers to write a new version of their program until they reach a version with the desired accuracy. As floating-point arithmetic is not intuitive and as there are many ways to write a program this process can be long and tedious.

Our work concerns the automatic optimization, at compile-time, of the accuracy of arithmetic expressions. To synthesize an accurate expression, we proceed in two phases. In the first phase, we build a large but yet polynomial under-approximation of all its mathematically equivalent expressions. In the second phase, we explore our abstract representation to find, still in polynomial-time, the expression with the best accuracy. More precisely, we select an expression which minimizes the roundoff errors in the worst case, i.e. for the worst inputs taken in the ranges specified by the user. This article mainly focuses on the first phase, the second phase not being described in details because of space limitations. Briefly speaking, this second phase uses an analysis *à la* Fluctuat to guide a local exploration of the abstract structure in order to extract an accurate expression. In this article, we present a new method to generate a large set of arithmetic expressions all mathematically equivalent. This kind of semantics-based transformation [3] has been introduced in [10, 11] and the current work strongly improves the existing transformations as it allows the generation of alternative expressions of very different shapes.

Technically, we define a intermediate representation called Abstract Program Expression Graph (APEG), presented in Section 2 and defined in Section 3, which is inspired from the Equivalence Program Expression Graphs (EPEG) introduced in [17]. Our APEGs are built thanks to a set of polynomial algorithms presented in Section 4. We have proven the correctness of our approach in Section 5, by introducing a Galois connection between sets of equivalent expressions and APEGs and we introduce an abstract semantics to under-approximate by APEGs the set of transformation traces of an arithmetic expression. We present in Section 6 an overview of how we extract an accurate expression from an APEG. Finally, Section 7 describes experimental results obtained with the Sardana tool which implements these techniques.

2 Overview

In this section, we give an overview of the methodology used to construct APEGs. APEGs are designed to represent, in polynomial size, many expressions that are *equal* to the original one we intend to optimize. Mathematical equality is defined with respect to a certain set $\triangleright$ of transformation rules of expressions, for example associativity and distributivity. Our goal is to build a tractable abstraction of

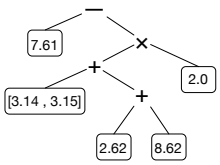

Fig. 1. Syntactic tree of expression e.

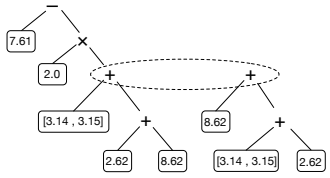

Fig. 2. APEG built on e by associativity.

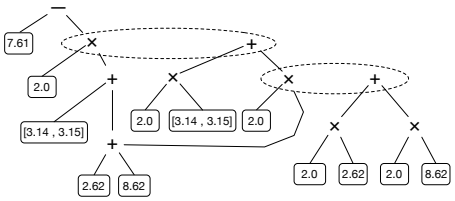

Fig. 3. Example of product propagation.

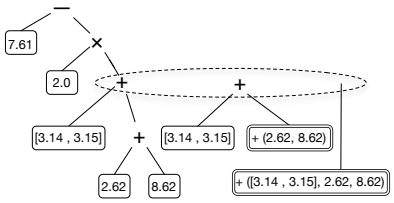

Fig. 4. APEG with abstraction boxes.

the set of equal expressions and then to explore this abstract set to find an expression which minimizes the roundoff errors arising during its evaluation.

First of all, an APEG is built upon the syntactic tree of an arithmetic expression. We assume that, for each input variable, an interval describing its range is provided by the user. An APEG then contains the usual arithmetic operators (like $+, \times$ or $-$), variables and constants in the interval domain. An example of syntactic tree is given in Figure 1 (intervals are written between brackets). An APEG has two main features: First, it is a compact data structure, of polynomial size, which is able to cope with the issue of a combinatorial explosion thanks to the concept of classes of *equivalent nodes*. Next, it contains *abstraction boxes* which represent an exponential number of expressions.

The first feature of APEGs is the notion of equivalent nodes. Equivalent nodes are obtained by attaching to each node of the tree a set of additional nodes (written inside dashed ellipses in the figures). An APEG is always built by adding new nodes in these sets of equivalent nodes, or by adding a new node with its own set of equivalent nodes. An important point is that nodes are never discarded. For example, if $\triangleright$ contains only the associativity of addition, we construct the APEG of Figure 2 over the expression $e = 7.61 - 2.0 \times ([3.14; 3.15] + (2.62 + 8.62))$. Remark that the APEG of Figure 2 represents the expressions $7.61 - 2.0 \times ([3.14; 3.15] + (2.62 + 8.62))$ and $7.61 - 2.0 \times (([3.14; 3.15] + 2.62) + 8.62)$ without duplicating the common parts of both expressions.

In order to produce various shapes of expressions, we use several algorithms to expand the APEG while keeping its size polynomial. First, by propagating the products in the APEG of Figure 2, we obtain the APEG of Figure 3. Next, we propagate the subtraction in products and sums. This transformation underlines the interest of APEGs: A naive approach would introduce a combinatorial

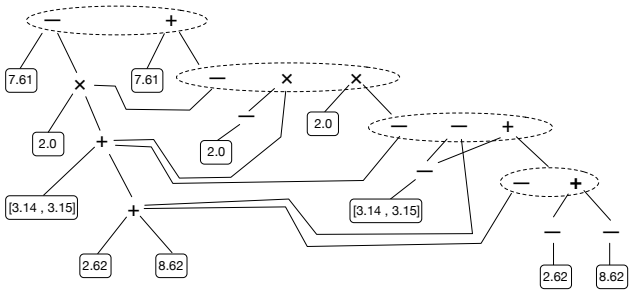

Fig. 5. Example of subtraction propagation.

explosion, since the propagation of a negation into each product can be done in two ways ($-(a \times b) = (-a) \times b = a \times (-b)$). Instead, as APEGs do not duplicate the common parts, we simply add to each multiplication a new branch connected to the lower part of the structure (see Figure 5). Thus we represent all the possible propagations of the subtraction without growing exponentially.

The second main feature of APEGs is the notion of abstraction box. We add abstraction boxes into APEGs in the sub-trees where the same operator is uniformly applied. Abstraction boxes are represented in our figures by rectangles with a double outline. Intuitively, an abstraction box is an abstraction of all the parsings that we can obtain with the sub-expressions contained in the box and a specific operator. For example, the box $\boxed{+, (a, b, c)}$ stands for any parsing of the sum of a, b and c. Abstraction boxes allow to represent exactly $(2n-1)!!$ [13, §6.3] equivalent expressions. An example of the abstraction boxes we add to the APEG of Figure 2 is given in Figure 4.

Our approach consists of combining all these transformations, in order to generate the largest (yet polynomial) APEG. The key idea is that we only add to APEGs expressions which are equivalent to the original one. The correctness relies on a Galois connection between a collecting semantics containing traces of transformation and evaluation of expressions and APEGs. This Galois connection is constructed as an under-approximation of the set of equivalent expressions in order to cover only equivalent expressions. Hence, we do not cover all the equivalent expressions but we represent an exponential number of them.

3 Formal Definition of APEGs

APEGs are inspired from the EPEG intermediate representation introduced in [17]. Initially, EPEGs were defined for the phase ordering problem, to represent multiple equivalent versions of an imperative program. They are built upon a C program by application of a set of rewriting rules until saturation. These rules correspond for example to constant propagations or loop unfoldings. This process is arbitrary stopped at a certain depth to avoid infinite processing. Our APEGs

are not built from a set of rewriting rules applied until saturation. Instead, we use a set of deterministic and polynomial algorithms described in Section 4.
An APEG is built from an initial expression e with respect to a certain set of binary relations $\triangleright = \{\triangleright_i, 1 \leq i \leq n\}$, representing the mathematically equivalent transformations we allow to perform on e. Usually we define $\triangleright$ as a subset of rules of the real field containing associativity, commutativity, distributivity and factorization. Formally, if an expression e_1 can be transformed into the expression e_2 using a relation of $\triangleright$, then e_1 and e_2 are mathematically equivalent. We generalize this property with the $\triangleright$-equal relation.

Definition 1. $\triangleright$-equal : *Let e_1 and e_2 be two arithmetic expressions, e_1 is $\triangleright$-equal to e_2 if $(e_1, e_2) \in \triangleright^\star$ where $\triangleright^\star$ is the transitive reflexive closure of the set of $\triangleright_i$ relations.*

APEGs are syntactic trees whose nodes are sets of $\triangleright$-equal expressions, and which contain abstraction boxes representing efficiently large sets of $\triangleright$-equal expressions. Abstraction boxes are defined by a binary symmetric operator $*$ (like $+$ or $\times$) and a set of operands L. Note that L may contain constants, variables, expressions or other abstraction boxes (abstraction boxes may be nested). The abstraction box $B = \boxed{*, L}$ represents the set of expressions made of the $*$ operator applied to the operands of L. For example, $\boxed{+, (x_1, x_2, x_3, x_4)}$ abstracts all the parsings of $\sum_{i=1}^{i=4} x_i$ and, for a nested box, $\boxed{+, (x_1, x_2, \boxed{+, (y_1, y_2, y_3)})}$ abstracts all the parsings of $\cup_{x_3 \in Y}\{\sum_{i=1}^{i=3} x_i\}$ where Y denotes all the parsings of $\sum_{i=1}^{i=3} y_i$.
Abstraction boxes are essential for our abstraction as they allow to represent efficiently an exponential number of $\triangleright$-equal expressions.

From a formal point of view, the set $\mathit{\Pi}_\triangleright$ of APEGs is defined inductively as the smallest set such that:

(i) $a \in \mathit{\Pi}_\triangleright$ where a is a leaf (a constant or an identifier or an interval $[x, y]$ abstracting all the values a such that $x \leq a \leq x$),
(ii) $*(lop, rop) \in \mathit{\Pi}_\triangleright$ where $*$ is a binary operator, lop and rop are APEGs representing the left and right operands of $*$,
(iii) $\boxed{*, (p_1, \ldots, p_n)} \in \mathit{\Pi}_\triangleright$ is an abstraction box defined by the operator $*$ and the APEGs $p_1, \ldots, p_n$ as operands,
(iv) $\langle p_1, \ldots, p_n \rangle \in \mathit{\Pi}_\triangleright$ is a class of $\triangleright$-equal expressions, where $p_1, \ldots, p_n$ are APEGS. Note that $p_1, \ldots, p_n$ cannot be classes of $\triangleright$-equal expressions themselves, i.e. $p_1, \ldots, p_n$ must be induced by the cases *(i)* to *(iii)* of the definition.

Case *(iv)* of the definition forbids nested equivalence classes since any equivalence class of the form $\langle p_1, \ldots, p_n, \langle p'_1, \ldots, p'_m \rangle \rangle$ could always be rewritten in $\langle p_1, \ldots, p_n, p'_1, \ldots, p'_m \rangle$. Examples of APEGs are given in figures 2 to 4. Equivalence classes are represented by dashed ellipses in the pictures.

4 APEG Construction

In this section, we introduce the transformations which add to APEGs new $\triangleright$-equal expressions and abstraction boxes. Each transformation is intended to only add new nodes into the APEGs without discarding any other node. First of all, recall from Section 3 that abstraction boxes are defined by a symmetric operator and a set of expressions. In order to produce the largest abstraction boxes, we have to introduce *homogeneous* parts inside APEGs.

Definition 2. Full homogeneity *Let $*$ be a symmetric binary operator and π an APEG. We say that π is fully homogeneous if it contains only variables or constants and the operator $*$.*
Partial homogeneity *We say that an APEG is partially homogeneous if it contains a fully homogeneous sub-expression $e_1 * \ldots * e_n$ where $\forall i, 1 \leq i \leq n, e_i$ is any sub-expression.*

For example, the expression $e = a+(b+c)$ is a fully homogeneous expression, while $e' = ((a \times b) + (c + d)) \times e$ is partially homogeneous since, for $e_1 = a \times b$ the sub-expression $e_1 + (c + d)$ of e' is fully homogeneous.

We introduce two kinds of transformations. First, we perform the homogenization of the APEG by adding new nodes which introduce new homogeneous sub-expressions. Next we apply the expansion functions which insert abstraction boxes in the homogenized APEGs. Both transformations are designed to be executed in sequence, in polynomial-time. The homogenization transformations insert into an APEG as many $\triangleright$-equal expressions as possible.

4.1 Homogenization Transformations

Transformation of multiplication Multiplication may yield two $\triangleright$-equal expressions: Either by applying the distributivity over the addition or subtraction, or by applying a further factorization to one or both of its operands (whenever it is possible). For example, the expression $e = a \times (b + c) + a \times d$ can be distributed either in $e_1 = (a \times b + a \times c) + a \times d$ or factorized into the expression $e_2 = a \times ((b + c) + d)$. In both cases e_1 and e_2 contain an homogeneous part for the $+$ operator. This transformation is illustrated in the upper part of Figure 6.

Transformation of minus The minus operator introduces three kinds of transformations depending on which expression it is applied to. If the minus operator is applied to an addition then it transforms the addition into a subtraction plus an unary minus operator. For example, $-(a+b)$ is transformed into $(-a)-b$. If the minus operator is applied to a multiplication then it generates two $\triangleright$-equal expressions, depending on the operands. For example, $-(a \times b)$ generates the $\triangleright$-equal expressions $(-a) \times b$ and $a \times (-b)$. If the minus operator is applied on another minus operator they anneal each other. This transformation is illustrated in the lower part of Figure 6. Note that, as shown in the graphical representation of the transformation given in Figure 6, in both cases (transformation of

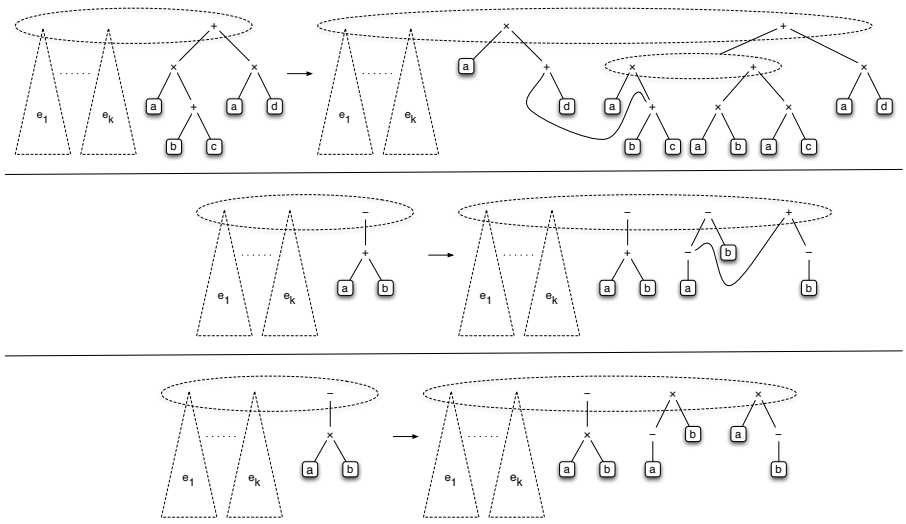

Fig. 6. Graphical representation of the homogenization transformations. $\triangleright$-equal expressions $e_1 \dots e_k$ are represented by dashed trees. The top transformation corresponds to the transformation over multiplication and the next two schemes illustrate the transformation over minus, for an addition and a product respectively.

multiplication and minus), we add as few nodes as possible to the pre-existing APEG. Each transformation only adds a polynomial number of node.

4.2 Expansion Functions

The expansion functions insert abstraction boxes with as many operands as possible. Currently, we have defined three expansion functions. From an algorithmic point of view, each expansion function is applied through all the nodes of the APEG, recursively. As the size of an APEG is polynomial in the number of its leaves, the expansion functions can be performed in polynomial-time.

Horizontal Expansion: The horizontal expansion introduces abstraction boxes which are built on some fully or partially homogeneous some sub-trees of an homogeneous part. If we split an homogeneous part in two, both parts are also homogeneous. Then we can either build an abstraction box containing the leaves of the left part of the homogeneous tree, or the leaves of the right part. For example let us consider the expression described in the top of Figure 7 where we perform and addition between the left sub-tree grouping the leaves $l_1, \dots, l_k$ and the right sub-tree grouping the leaves $l'_1, \dots, l'_{k'}$. We can either create a box $B_1 = (\boxed{+, (l_1, \dots, l_k)})$ or a box $B_2 = (\boxed{+, (l'_1, \dots, l'_{k'})})$. In one case we collapse all the parsings of $\sum_{i=1}^{k} l_i$ and keep a certain parsing of $\sum_{j=1}^{k'} l'_j$ (in an englobing expression). In the other case we keep a certain parsing of $\sum_{i=1}^{k} l_i$ plus

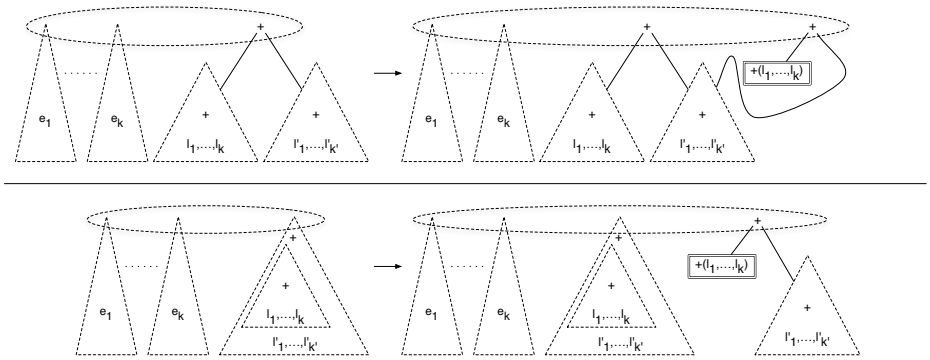

Fig. 7. Graphical representation of the expansion transformations. The dotted triangles with $l_1, \ldots, l_k$ written inside represent homogeneous parts. From top to bottom, the figure represents the horizontal and vertical expansion transformations.

any parsing of $\sum_{j=1}^{k'} l'_j$. This transformation is illustrated in Figure 7. We introduce only $O(2n)$ boxes, among the exponential number of possible combinations.

Vertical Expansion: The vertical expansion introduces abstraction boxes in an homogeneous structure by splitting it into two parts. Here, the splitting is performed by considering in one hand the leaves contained in a sub-expression and in the other hand the leaves contained in the englobing expression. Let us consider an homogeneous structure defined by a set $P = \{p_1, \ldots, p_n\}$ of operands and the binary operator $*$. Each occurrence of $*$ defines a sub-expression with a set $\{p'_1, \ldots, p'_k\} \subseteq P$ of leaves. The vertical expansion introduces for each occurrence of $*$, an abstraction box defined by $*$ and the set $P \setminus \{p'_1, \ldots, p'_k\}$ of leaves. Also the vertical expansion introduces for each leaf an abstraction box containing all the others. This transformation is illustrated in Figure 7. It introduces $O(2n)$ boxes into an homogeneous part of size n.

Box expansion: The box expansion is designed to add new abstraction boxes over the existing ones. As we allow abstraction boxes to be recursive, then for any abstraction box $B' = (\boxed{*', P'})$ which is contained in $B = (\boxed{*, P})$, if $* = *'$ then we can merge P and P' into a new abstraction box $B'' = (\boxed{*, P \cup P'})$. It is obvious that $\rhd$-equal expressions represented by B'' strictly includes the $\rhd$-equal expressions represented by B.

5 Correctness

5.1 Collecting Semantics

For the sake of clarity, we define a collecting semantics enabling only the transformation of expressions and we omit to include the reduction rules corre-

sponding to the usual evaluation of expressions. Let $(\!|e|\!)_{\rhd}$ be the set of partial traces for the transformation of e into $\rhd$-equal expressions. To define this collecting semantics, we need to introduce some transformation rules of arithmetic expressions into other equivalent expressions. We define $\mathcal{R} = \cup_{i=1}^{n} \rhd_i$ with $\forall i,\ 1 \leq i \leq n,\ \rhd_i \subseteq Expr \times Expr$. We do not require the $\rhd_i$ relations to be transitive since $\rhd$ may be applied many times along a trace. For example, we can set $\rhd_1 = \{((a+b)+c,\ a+(b+c)) \in Expr^2 : a,b,c \in Expr\}$, $\rhd_2 = \{((a+b) \times c,\ a \times c + b \times c) \in Expr^2 : a,b,c \in Expr\}$ and $\rhd_3$ and $\rhd_4$ the symmetric relations of $\rhd_1$ and $\rhd_2$. We define the transformation relation $\rhd$ by means of the rules below, where $*$ stands for $+$, $-$ or $\times$:

$$\frac{e \rhd_i e',\ \rhd_i \in \mathcal{R}}{e \rhd e'} \qquad \frac{e_1 \rhd e_1'}{e_1 * e_2 \rhd e_1' * e_2} \qquad \frac{e_2 \rhd e_2'}{e_1 * e_2 \rhd e_1 * e_2'} \tag{1}$$

Next we define $(\!|e|\!)_{\rhd}$ as the set $\rhd^*$ of $\rhd$-chains, i.e. the set of all the sequences $e \rhd e_1 \rhd \ldots \rhd e_n$ such that $\forall i,\ 1 \leq i < n$, $e_i \in Expr$ and $e_i \rhd e_{i+1}$ and $e \rhd e_1$.

Obviously, the collecting semantics $(\!|e|\!)_{\rhd}$ is often intractable on a computer. For example the number of $\rhd$-equal expressions is exponential if $\rhd$ contains the usual laws of the real field (associativity, distributivity, etc.) Our abstraction of the collecting semantics by APEGs is an *under-approximation*. We compute our APEG abstract value by iterating a function $\Phi : \Pi_{\rhd} \to \Pi_{\rhd}$ until a fixed point is reached: $[\![e]\!]^\sharp = Fix\ \Phi(\bot)$. The function Φ corresponds to the transformations introduced in Section 4. The correctness stems from the fact that we require that a) Φ is extensive, ie. $\forall t^\sharp \in \Pi_{\rhd},\ t^\sharp \sqsubseteq \Phi(t^\sharp)$, b) Φ is Scott-continuous (ie. $x \sqsubseteq y \Rightarrow \Phi(x) \sqsubseteq \Phi(y)$ and for any increasing chain X, $\sqcup_{x \in X} \Phi(x) = \Phi(\sqcup X)$) and c) for any set of abstract traces $t^\sharp$, $\gamma(t^\sharp) \subseteq (\!|e|\!)_{\rhd} \Rightarrow \gamma(\Phi(t^\sharp)) \subseteq (\!|e|\!)_{\rhd}$. These conditions holds for the transformations of Section 4 which only add $\rhd$-equal elements in APEGs and never discard existing elements. By condition a), the chain C made of the iterates $\bot$, $\Phi(\bot)$, $\Phi^{(2)}(\bot)$, ... is increasing. Then C has an upper bound since $\Pi_{\rhd}$ is a CPO (see Section 5.3). The function Φ being continuous, $\sqcup_{c \in C} \Phi(c) = \Phi(\sqcup C)$ and, finally, by condition c) $\gamma([\![e]\!]^\sharp) = \gamma(Fix\ \Phi(\bot)) = \gamma(\sqcup_{c \in C} \Phi(c)) = \gamma(\Phi(\sqcup C)) \sqsubseteq (\!|e|\!)_{\rhd}$.

Intuitively, computing an under-approximation of the collecting semantics ensures that we do not introduce into the APEG some expressions that would not be mathematically equivalent to e using the relations in $\rhd$. This is needed to ensure the correctness of the transformed expression. Using our conditions, any abstract trace of the resulting APEG is mathematically correct wrt. the transformation rules of $\rhd$ and can be chosen to generate a new expression.

5.2 Abstraction and Concretization Functions

For an initial expression e, the set $(\!|e|\!)_{\rhd}$ contains transformations of the expression e into $\rhd$-equal expressions as defined in Equation (1). The elements of $(\!|e|\!)_{\rhd}$ are of the form $e \rhd e' \rhd \ldots \rhd e^n$, where $e, e', \ldots, e^n$ are $\rhd$-equal and we may aggregate them into a global APEG since this structure has been introduced to represent multiple $\rhd$-equal expressions. So we define the abstraction function

α, as the function that aggregates each expression contained in the traces in a single APEG. In order to define the concretization function γ we introduce the following functions:

- the function $\mathcal{C}(p, \pi)$ which returns the set of sub-APEGs of π which are in the same equivalence class than p, In other words, $\mathcal{C}(p, \pi) = \{p_1, \dots p_n\}$ if there exists an equivalence class $\langle p_1, \dots p_n \rangle$ in π such as $p \in \langle p_1, \dots p_n \rangle$,
- the composition $\circ_*$ of two traces by some operator $*$. Intuitively, given evaluation traces t_1 and t_2 for two expressions e_1 and e_2, we aim at building the evaluation trace of $e_1 * e_2$. Following the rules of Equation (1), $\circ_*(t_1, t_2)$ is the trace in which, at each step, one of the sub-expressions e_1 or e_2 of $e_1 * e_2$ is transformed as they were transformed in t_1 or t_2.

The concretization γ of an APEG $\pi \in \Pi_\rhd$ is defined by induction by:

(i) if $\pi = a$ where a is a leaf (i.e. a constant or a variable) then $\gamma(\pi) = \{a\}$,

(ii) if $\pi = *(lop, rop)$ where $*$ is a binary operator, and lop and rop are the operands of $*$, if the traces of $\gamma(\mathcal{C}(lop, \pi))$ are of the form $t = t_0 \rhd \dots \rhd t_n$, and the traces of $\gamma(\mathcal{C}(rop, \pi))$ are of the form $s = s_0 \rhd \dots \rhd s_m$, then we have

$$\gamma(*(lop, rop)) = \bigcup_{\substack{t \in \gamma(\mathcal{C}(lop,\pi)), |t| = n \\ s \in \gamma(\mathcal{C}(rop,\pi)), |s| = m}} t_0 * s_0 \rhd t_1 * s_1 \rhd \dots \rhd t_{n+m} * s_{n+m} \tag{2}$$

where at each step either $t_i \rhd t_{i+1}$ and $s_i = s_{i+1}$, or $t_i = t_{i+1}$ and $s_i \rhd s_{i+1}$, and where $|t|$ is the length of the trace t.

(iii) if $\pi = \langle p_1, \dots, p_n \rangle$, let us take p_i and p_j, two distinct nodes in π. Let $t \in \gamma(p_i)$ and $t' \in \gamma(p_j)$ such as $t = t_0 \rhd \dots \rhd t_n$ and $t' = t'_0 \rhd \dots \rhd t'_m$. We defined $\mathcal{J}_{ij}$ the set of all pairs (k, l) with $0 \leq k \leq n$ and $0 \leq l \leq m$ such as $t_k \rhd t'_l$ is a valid transformation. Then we defined $\gamma(\pi)$ as all the $\rhd$-compatible junction of pieces of traces of $\gamma(pi)$ and $\gamma(p_j)$ for all p_i and p_j. Formally

$$\gamma(\pi) = \bigcup_{\substack{p_i, p_j \in \pi \\ (k,l) \in \mathcal{J}_{ij}}} t_0 \rhd \dots \rhd t_k \rhd t'_l \rhd \dots \rhd t_m \tag{3}$$

This definition works for one function point between two traces, but it could be generalized to multiple junction points.

(iv) if $\pi = \boxed{*, (p_1, \dots, p_2)}$ then, by definition of an abstraction box, $\gamma(\pi) = \bigcup_{p \in P} \gamma(p)$, where P is the set of all the parsing of $p_1, \dots, p_n$ using the binary operator $*(lop, rop)$ whose concretization is defined in Point *(ii)*.

5.3 The Abstract Domain of APEGs

In this section, we show that the set of APEGs is a complete partial order. Then we show the existence of a Galois connection between sets of traces and APEGs.

First, we define $\sqsubseteq_\square$, the partial order on the set of abstraction boxes. Let $B_1 = \boxed{*, (p_1, \ldots, p_n)}$ and let $B_2 = \boxed{*', (p'_1, \ldots, p'_m)}$, we say that $B_2 \sqsubseteq_\square B_1$ if and only if the following conditions are fulfilled:

(i) $* = *'$,
(ii) $\forall p'_i \in \{p'_1, \ldots, p'_m\}$, if p'_i is not an abstraction box, $\exists p_j \in \{p_1, \ldots, p_n\}$ such that $p_j = p'_i$,
(iii) $\forall p'_i \in \{p'_1, \ldots, p'_m\}$, if p'_i is an abstract box $B_3 = \boxed{*'', (p''_1, \ldots, p''_k)}$ we have:
  (a) if $*'' = *$ then $\forall p''_j \in \{p''_1, \ldots, p''_k\}$ if p''_j is not an abstraction box then $p''_j \in \{p_1, \ldots, p_n\}$, else if p''_j is an abstract box then $\exists p_i \in \{p_1, \ldots, p_n\}$ such that p_i is an abstract box and $p''_j \sqsubseteq_\square p_i$,
  (b) if $*'' \neq *$ then $\exists p_j \in \{p_1, \ldots, p_n\}$ such that p_j is an abstraction box and $p'_i \sqsubseteq_\square p_j$.

In order to define the join $\sqcup_\square$ of two boxes $B_1 = \boxed{*, (p_1, \ldots, p_n)}$ and $B_2 = \boxed{*', (p'_1, \ldots, p'_m)}$, we introduce $B_3 = \boxed{*, (p_1, \ldots, p_n, p'_1, \ldots, p'_m)}$. By definition, $B_1 \sqcup_\square B_2 = B_3$ if $* = *'$, otherwise, if $* \neq *'$ then $B_1 \sqcup_\square B_2 = \top$. Next we extend the operators $\sqsubseteq_\square$ and $\sqcup_\square$ to whole APEGs. We obtain new operators $\sqsubseteq$ and $\sqcup$ defined as follows. For $\sqsubseteq$, given two APEGs $\pi_1, \pi_2 \in \Pi_\triangleright$ we have $\pi_1 \sqsubseteq \pi_2$ if and only if one of the following conditions hold:

(i) $\pi_1 = a$, $\pi_2 = a'$ and $a = a'$, where a is a constant or an identifier,
(ii) if π_1 and π_2 fulfill all of the following conditions: $\pi_1 = *(lop, rop)$, $\pi_2 = *'(lop', rop')$, $* = *'$, $lop \sqsubseteq lop'$ and $rop \sqsubseteq rop'$,
(iii) if $\pi_1 = \langle p_1, \ldots, p_n \rangle$, $\pi_2 = \langle p'_1, \ldots, p'_m \rangle$ and $\forall i, 1 \leq i \leq n, \exists j, 1 \leq j \leq m$ such that $p_i \sqsubseteq p'_j$,
(iv) if π_1 is a fully homogeneous APEG defined by $*$ and the nodes $\{p_1, \ldots, p_n\}$, and π_2 contains an abstraction box B' such that $\boxed{*, (p_i, ..., p_n)} \sqsubseteq_\square B'$,
(v) if $\pi_1 = \langle p_1, \ldots, p_n \rangle$, $\pi_2 = *(lop, rop)$, $lop \in \langle p^l_1, \ldots, p^l_{k_l} \rangle$, $rop \in \langle p^r_1, \ldots, p^r_{k_r} \rangle$ and $\forall p_i \in \pi_1, \exists p^l_j \in \mathcal{C}(lop, \pi)$ and $\exists p^r_k \in \mathcal{C}(rop, \pi)$ such that $p_i \sqsubseteq *(p^l_j, p^r_k)$.

In order to define $\pi_1 \sqcup \pi_2$, with $\pi_1, \pi_2 \in \Pi_\triangleright$, we observe first that π_1 and π_2 only contain $\triangleright$-equal expressions. The join of two APEGs π_1 and π_2 is defined as the union of the corresponding trees. Boxes are joined using $\sqcup_\square$ and the join of two nodes of the syntactic tree p_1 and p_2 yields the equivalence class $\langle p_1, p_2 \rangle$. Finally we define $\bot$ as the empty APEG, and $\top$ as the APEG built with all the possible expression transformations of $\triangleright$.
We have the following Galois connection between the collecting semantics and the APEGs where $\wp(X)$ denotes the powerset of X:

$$\langle \wp((\!|e|\!)_\triangleright), \subseteq \rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle \Pi_\triangleright, \sqsubseteq \rangle \tag{4}$$

6 Profitability Analysis

In this section we give an overview of how our profitability analysis works. First, we recall how the roundoff errors are computed, and next we briefly describe the search algorithm employed to explore APEGs.

We use a non-standard arithmetic where error terms are attached to the floating-point numbers [1, 9, 11]. They indicate a range for the roundoff error due to the rounding of the exact value in the current rounding mode. The exact error term being possibly not representable in finite precision, we compute an over-approximation and return an interval with bounds made of multiple precision floating-point numbers. Indeed, the error interval may be computed in an arbitrarily large precision since it aims at binding a real number and, in practice, we use the GMP multi-precision library [18]. Note that the errors can be either positive or negative. This depends on the direction of the rounding operation which can create either an upper or a lower approximation.

Error terms are propagated among computations. The error on the result of some operation $x * y$ is the propagation of the errors on x and y through the operator $*$ plus the new error due to the rounding of the result of the operation itself. Let x and y be to values represented in our arithmetic by the pairs (f_x, e_x) and (f_y, e_y) where f_x and f_y are the floating-point or fixed-point numbers approximating x and y and e_x and e_y the error terms on both operands. Let $\circ(v)$ be the rounding of the value v in the current rounding mode and let $\varepsilon(v)$ be the roundoff error, i.e. the error arising when rounding v into $\circ(v)$. We have by definition $\varepsilon(v) = v - \circ(v)$ and, in practice, when v is an interval, we approximate $\circ(v)$ by $[-\frac{1}{2}ulp(m), \frac{1}{2}ulp(m)]$ in floating-point arithmetic, or by $[0, ulp(m)]$ in fixed-point arithmetic, where m is the maximal bound of v, in absolute value, and ulp is the function which computes the unit in the last place of m [14]. The elementary operations are defined in equations (5) to (7).

$$x + y = \big(\circ(f_x + f_y), e_x + e_y + \varepsilon(f_x + f_y)\big) \tag{5}$$

$$x - y = \big(\circ(f_x - f_y), e_x - e_y + \varepsilon(f_x - f_y)\big) \tag{6}$$

$$x \times y = \big(\circ(f_x \times f_y), f_y \times e_x + f_x \times e_y + e_x \times e_y + \varepsilon(f_x \times f_y)\big) \tag{7}$$

For an addition, the errors on the operands are added to the error due to the roundoff of the result. For a subtraction, the errors on the operands are subtracted. The semantics of the multiplication comes from the development of $(f_x + e_x) \times (f_y + e_y)$. For other operators, like division and square root, we use power series developments to compute the propagation of errors [9].

We use the former semantics to evaluate which expression in an APEG yields the smallest error. The main difficulty is that it is possible to extract an exponential number of expressions from an APEG. For example, let us consider an operator $*(p_1, p_2)$ where p_1 and p_2 are equivalence classes $p_1 = \langle p'_1, \dots p'_n \rangle$ and $p_2 = \langle p''_1, \dots p''_m \rangle$. Then we have to consider all the expressions $*(p'_i, p''_j)$ for $1 \leq i \leq n$ and $1 \leq j \leq m$. In general, the sub-APEGs contained in p_1 and p_2 may be operations whose operands are again equivalence classes. To cope with this combinatorial explosion, we use a limited depth search strategy. We select the way an expression is evaluated by considering only the best way to evaluate its sub-expressions. This corresponds to a local choice. In our example, synthesizing an expression for $*(p_1, p_2)$ consists of searching the expression $p'_i * p''_j$ whose error is minimal with respect to any $p'_i \in p_1$ and any $p''_j \in p_2$.

Table 1. Statistical improvement of accuracy for summation and polynomials.

expression form	interval width	10 terms expression		20 terms expression	
		large	small	large	small
100%+	Configuration 1	35.3%	33.4%	16.2%	16.5%
	Configuration 2	35.2%	34.3%	15.8%	34.3%
	Configuration 3	54.2%	59%	46.5%	51.9%
	Configuration 4	46.2%	52.9%	41.4%	46.3%
45%+, 10%×, 45%−	Configuration 1	12.9%	14.5%	13.1%	15%
	Configuration 2	11.8%	12.9%	11.8%	12%
	Configuration 3	15.1%	14.9%	13.9%	14.5%
	Configuration 4	10.0%	11.3%	11%	11.4%
50%+, 25%×, 25%−	Configuration 1	15%	16.4%	15.2%	16.4%
	Configuration 2	12.9%	13.6%	12.2%	13.1%
	Configuration 3	18.4%	17.7%	16.4%	16.9%
	Configuration 4	12.7%	13.5%	12.2%	12.3%

For a box $B = \boxed{*, (p_1, \ldots, p_n)}$ we use an heuristic which synthesizes an accurate expression (yet not always optimal). This heuristic is defined as a greedy algorithm which searches at each step the pair p_i and p_j such that the error term carried out by the expression $p_i * p_j$ is minimal. Then p_i and p_j are removed from the box and a new term p_{ij} is added whose accuracy is equal to the error term of $p_i * p_j$ defined by Equations (5) to (7). This process is repeated until there is only one node left in the box. This last node corresponds to the root of the expression synthesized for the abstraction box. Remark that other algorithms could be used including algorithms performing additional computations to compensate the errors [19, 16].

7 Experimental results

In this section, we present experimental results obtained using our tool, Sardana. We present statistical results on randomly generated expressions. Then we show exhaustive tests on summations and polynomial functions.

7.1 Statistical Results

In this section, we present statistical results concerning the reduction of the roundoff errors on randomly generated expressions. First, we consider summations whose operands belong to intervals. Summations are fundamental in our domain since they correspond to the core of many numerical algorithms (scalar products, matrix products, means, integrators, etc). Despite their apparent simplicity, summations may introduce many accuracy errors and many algorithms have been proposed (this is still an active research field e.g. [19]). Hence, a main challenge for our analysis is to improve the accuracy of sums.

We use 4 configurations taken from [8] and which illustrate several pitfalls of the summation algorithms in floating-point arithmetic. We call *large value* a floating-point interval around 10^{16}, *medium value* an interval around 1, and *small value* an interval around 10^{-16}. We consider the following configurations:

1) Only positive sign, 20% of large values among small values. Accurate sums should first add the smallest terms,
3) Only positive sign, 20% of large values among small and medium values. Accurate sums should add terms in increasing order,
3) Both signs, 20% of large values that cancel, among small values. Accurate sums should add terms in decreasing order of absolute values,
4) Both signs, 20% small values and same number of large and medium values. Accurate sums should add terms in decreasing order of absolute values.

For all these configurations, we present in the first row of Table 1 the average improvement on the error bound, i.e. the percentage of reduction of the error bound. We test each configuration on two expression sizes: With 10 or 20 terms, and with two widths of intervals: Small width (interval width about 10^{-12} times the values) or large width (interval width about 10% of the values). Each result is an average of the error reduction on 10^3 randomly generated expressions. Each source expression has been analyzed in the IEEE-754 binary 64 format by our tool in matter of milliseconds on a laptop computer. We can see that our tool is able to reduce the roundoff error on the result by 30% to 50% for a 10 terms, and between 16% and 45% for 20 terms. This means that our tool synthesize new expressions whose evaluation yields smaller roundoff errors than the original ones in the worst case, for any concrete configuration taken into the intervals for which the transformation has been performed.

Table 1 presents also the average improvement for more complex expressions. We used the same configurations as before but on two new sets of randomly generated expressions: The former with 45% of sums, 10% of products and 45% of subtractions, and the latter with 50% of additions, 25% of products and 25% subtractions. We obtained an accuracy improvement by 10% to 18% in average. We believe that the accuracy improvement is less significant because the data are not specifically ill-conditioned for these kind of expressions.

7.2 Benchmarks

Transformation of Summations First we present some exhaustive tests concerning the summations. Our goal is to determine the performance of our tool for all the possible initial parsings of a sum. Let us remark that a sum of n terms has $(2n-1)!!$ evaluation schemes [13, §6.3] which can all have various accuracies. For example, a 9 term sum yields 2 millions evaluation schemes, and a 10 term sum yields almost 40 millions schemes. We performed our benchmarks on all the initial parsings of sums going from 5 to 9 terms. For each parsing we have tested the same four configurations and the two interval widths of value described in Section 7.1. We present the results obtained using the IEEE-754 binary 64 format

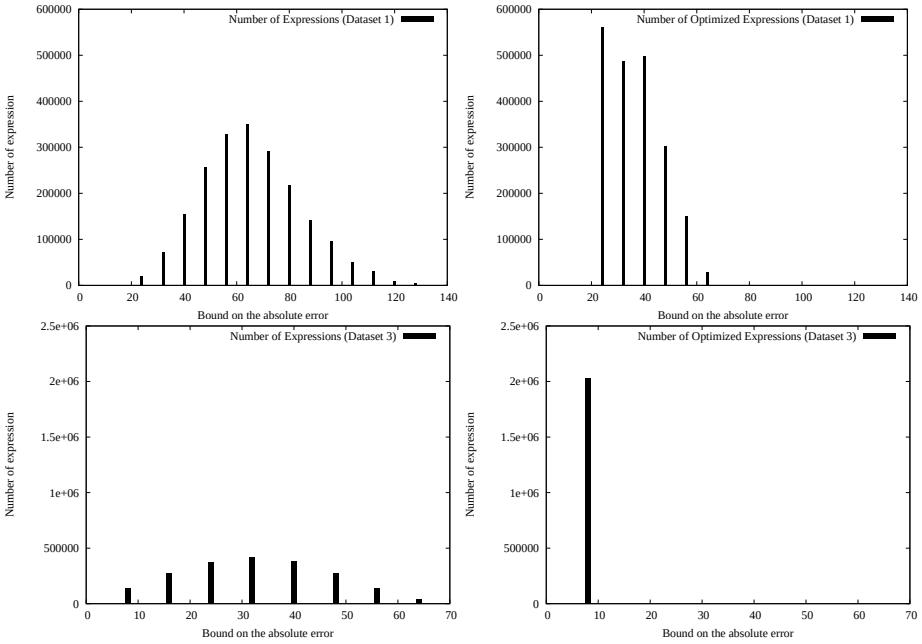

Fig. 8. *First line: Results for the sum of 9 terms, Configuration 1. Second line: Results with Configuration 2. Left and right part illustrate the initial and optimized accuracy.*

to perform 9 terms summations with large interval width (other interval widths yield similar observations and this configuration presents the most significant results of our benchmarks).

Our results are depict by histograms organized as follows: The x-axis indicates the roundoff error on the result of the evaluation of one summation (i.e. for a specific parsing) using the configuration mentioned in the caption and the y-axis indicates how many parsings among all the initial parsings have introduced the corresponding roundoff error (note that many parsings yield the same error, for instance about $3,5 \cdot 10^6$ yield an absolute error of magnitude 64 in the first histogram of figure 8). We have first performed an analysis to determine the maximal error bound on each source sum. This corresponds to the leftmost histograms of Figures 8 and 9. Then we have applied our tool to each source sum with the same data in order to obtain the error bounds on the optimized sums. The right-hand side of Figure 8 gives the number of sums corresponding to each accuracy after program transformation. Intuitively, the more the bars are shifted to the left, the better it is. In all the figures presented in this section, both the leftmost and rightmost histograms of each line have the same scale.

First, let us remark that, initially, the distribution of the errors is similar for each configuration (leftmost histograms of figures 8 and 9). The distribution looks like gaussian: There are few optimal parsings (leftmost bar) and few parsings returning the worst accuracy (rightmost bar). Remark that on config-

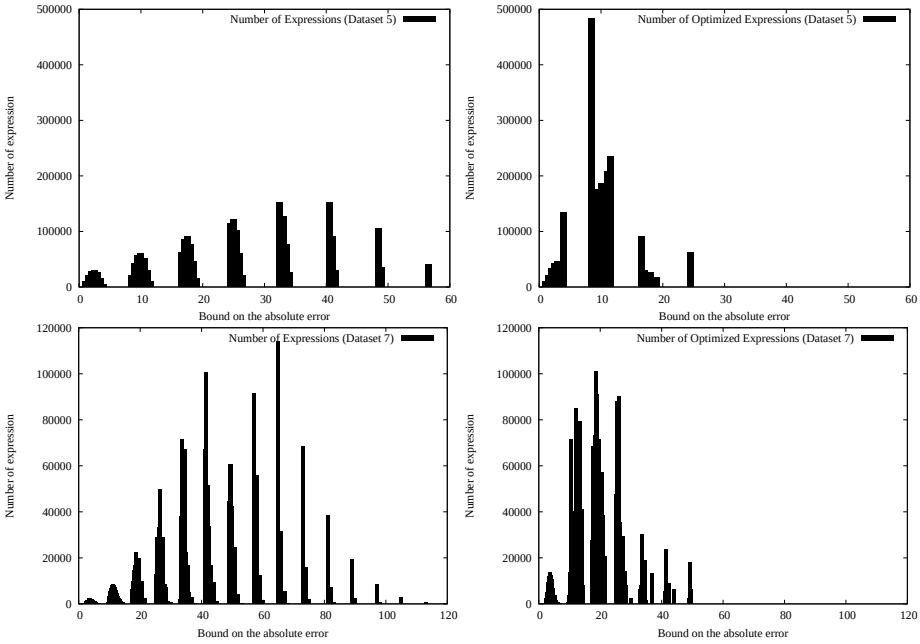

Fig. 9. *Sum of 9 terms for configurations 3 and 4 (first and second line resp.)*

urations 1, 3 and 4 our tool is able to shift the gaussian-like distribution of the bars to the left, which corresponds to an average gain of 50% of accuracy.

For each sum in Configuration 2 our tool is able to produce a parsing of optimal accuracy. This result is due to how we generate code when we reify an abstraction box: We perform a greedy association of terms and, in the case of positive values, it corresponds to sorting them by increasing order of magnitude which is the optimal solution in this case.

Transformation of Polynomials We focus now on the transformation of monovariate polynomials. Polynomials are pervasives in numerical codes yet it is less famous that numerical errors arise during their evaluation close to a root (and even more close to a multiple root [7]). We have tested exhaustively all the polynomials defined by $P^n(x) = \sum_{k=0}^{n} (-1)^k \times \binom{n}{k} \times x^k$which correspond to the developed form of the function $(x-1)^n$. In our source expressions, x^n is written as the product $\prod_{i=1}^{n} x$. We let n range from 2 to 5. The variable x is set to an interval around 1 ± 10^{-12} in the IEEE-754 binary 64 format. To grasp the combinatorial explosion in the number of ways to evaluate the polynomial, note that for $n = 5$ there are 2.3 million distinct schemes, and for $n = 6$ there are 1.3 billion schemes [13, §6.2.2]. Left part of Figure 10 shows the error distribution of the initial schemes of $P^5(x)$ and the right part shows the error distribution of the optimized schemes. We can see that initially most of the schemes induces a

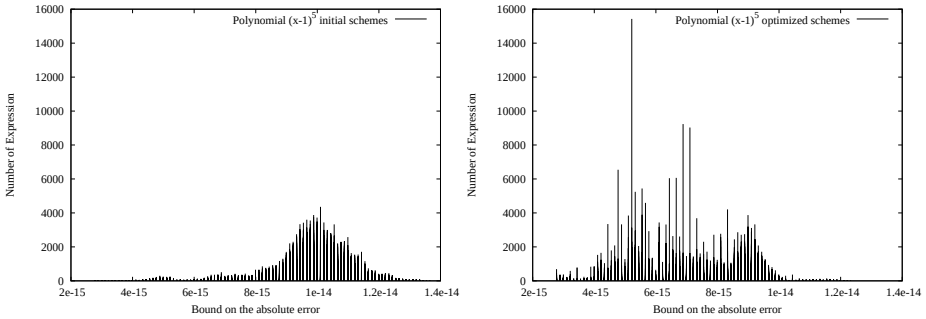

Fig. 10. *Leftmost histogram illustrates the initial accuracy of the polynomials $P^5(x)$, the rightmost histogram yields the accuracy of the optimized one.*

rounding error which is between $8.0 \cdot 10^{-15}$ and $1.2 \cdot 10^{-14}$. Our tool produces optimized schemes of $P^5(x)$ with an error bound between $4.0 \cdot 10^{-15}$ and $9.0 \cdot 10^{-15}$, which represents a 25% to 50% improvement of the numerical accuracy.

8 Conclusion

In this article, we have introduced a new technique to represent a large set of mathematically equal arithmetic expressions. Our goal is to improve the numerical accuracy of an expression in floating-point arithmetic. We have define an abstract intermediate representation called APEG which represents very large set of arithmetic expressions that are equal to an original one. We construct APEGs by using only deterministic and polynomial functions, which allow us to represent an exponential number of equal expressions of very various shapes. The correctness is based on a Galois connection between the collecting semantics of transformations of arithmetic expressions and our abstract domain of APEG. Our experimental results show that, statistically, the roundoff error on summations may be reduced by 40% to 50% and by 20% for polynomials. We intend to present in more details the approach we use to explore APEGs and select expressions, as well as the implementation of our tool.

We believe that our method can be improved and extended in many ways. First, we want to introduce more expansion functions in order to increase the variety of equal expressions in APEGs. We already think about defining some expansion functions to achieve partial regroupings of identical terms in a sum. Then we want to extend APEGs in order to handle the transformation of whole pieces of programs and not only isolated arithmetic expressions. We intend to handle control structure as well as recursive definitions of variables or iteration structure. At short term, we aim at transforming small standalone programs such as embedded controllers or small numerical algorithms.

9 Acknowledgments

We would like to thank Radhia and Patrick Cousot, Damien Massé and all the members of the Abstraction team for helpful discussions on the formalization of our transformation as the under-approximation of a collecting semantics.

References

1. ANSI/IEEE. *IEEE Standard for Binary Floating-point Arithmetic*, std 754-2008 edition, 2008.
2. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximations of fixed points. In *POPL*, pages 238–252. ACM, 1977.
3. P. Cousot and R. Cousot. Systematic design of program transformation frameworks by abstract interpretation. In *POPL*, pages 178–190. ACM, 2002.
4. D. Delmas, E. Goubault, S. Putot, J. Souyris, K. Tekkal, and F. Vedrine. Towards an industrial use of fluctuat on safety-critical avionics software. In *Formal Methods for Industrial Critical Systems, 14th International Workshop*, FMICS, 2009.
5. E. Goubault, M. Martel, and S. Putot. Static analysis-based validation of floating-point computations. In *Numerical Software with Result Verification*, number 2991 in LNCS, pages 306–313, 2004.
6. E. Goubault and S. Putot. Static analysis of finite precision computations. In *VMCAI*, number 6538 in LNCS, pages 232–247, 2011.
7. N. J. Higham. *Accuracy and Stability of Numerical Algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, second edition, 2002.
8. P. Langlois, M. Martel, and L. Thévenoux. Accuracy Versus Time: A Case Study with Summation Algorithms. In *PASCO*, pages 121–130. ACM, 2010.
9. M. Martel. Semantics of roundoff error propagation in finite precision calculations. *Journal of Higher Order and Symbolic Computation*, 19:7–30, 2006.
10. M. Martel. Semantics-based transformation of arithmetic expressions. In *Static Analysis Symposium, SAS*, number 4634 in LNCS. Springer, 2007.
11. M. Martel. Enhancing the implementation of mathematical formulas for fixed-point and floating-point arithmetics. *Journal of Formal Methods in System Design*, 35:265–278, 2009.
12. D. Monniaux. The pitfalls of verifying floating-point computations. *TOPLAS*, 30(3):12, 2008.
13. C. Mouilleron. *Efficient computation with structured matrices and arithmetic expressions*. PhD thesis, Université de Lyon–ENS de Lyon, November 2011.
14. J.-M. Muller. On the definition of ulp(x). Technical Report 5504, INRIA, 2005.
15. J.-M. Muller, N. Brisebarre, F. de Dinechin, C.-P. Jeannerod, V. Lefèvre, G. Melquiond, N. Revol, D. Stehlé, and S. Torres. *Handbook of Floating-Point Arithmetic*. Birkhäuser Boston, 2010.
16. T. Ogita, S. M. Rump, and S. Oishi. Accurate sum and dot product. *SIAM Journal on Scientific Computing (SISC)*, 26(6):1955–1988, 2005.
17. R. Tate, M. Stepp, Z. Tatlock, and S. Lerner. Equality saturation: A new approach to optimization. In *POPL*, pages 264–276. ACM, 2009.
18. Torbjorn Granlund and the GMP development team. *The GNU Multiple Precision Arithmetic Library*, 5.0.2 edition, 2011. `http://gmplib.org`.
19. Y.-K. Zhu and W. B. Hayes. Algorithm 908: Online exact summation of floating-point streams. *Transactions on Mathematical Software*, 37(3):1–13, 2010.