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

Arnault Ioualalen and Matthieu Martel

DALI - Université de Perpignan Via Domitia
52 avenue Paul Alduy
66860 Perpignan Cedex 9, France

LIRMM
CNRS: UMR 5506 - Université Montpellier 2
161 rue Ada
34095 Montpellier Cedex 5, France

ABSTRACT

Floating-point arithmetic is an important source of errors in programs because of the loss of accuracy it introduces. Even if we can bound statically the roundoff error due to the evaluation of an arithmetic expression, it is still difficult to transform this expression into a more accurate one. The goal of this article is to introduce a new semantics-based transformation to automatically replace arithmetic expressions by other expressions which are mathematically equal but whose evaluation in the floating-point arithmetic introduces smaller roundoff errors. The main difficulty is that there is in general an exponential number of mathematically equal expressions, up to associativity, commutativity and distributivity. Our approach is based on abstract interpretation and we introduce a new abstract domain to represent in polynomial size an exponential number of mathematically equivalent expressions. Compared to previous approaches, this technique strongly improves the variety of expressions that we can capture in the abstract and makes it possible to distinguish mathematically equal expressions of very different shapes.

1. INTRODUCTION

In computers, exact computations are approximated using the floating-point arithmetic which relies on a finite representation of the numbers [1, 13]. Although this approximation is accurate enough for most applications, in some cases, the results become irrelevant or too inaccurate with respect to the needs of the user. In programs, these roundoff errors are very difficult to understand and to rectify by hand. At least this task is strongly time consuming and, often, it is almost impossible.

Recently, validation techniques based on abstract interpretation [2] have been developed to assert the numerical accuracy of these computations and to help the programmer to correct their codes [8]. For example, Fluctuat is a static analyzer that enables one to detect the inaccuracies in C codes and to understand their origin [6, 4]. This tool has already been successfully used in many industrial projects, in aeronautics and other industries. However, this method does not indicate how programs have to be corrected in order to produce smaller errors. The programmers have to repeatedly write new versions of their codes and re-analyze until they reach a version which meets the desired accuracy. This process can be long and tedious as there are many ways to write a program and as understanding errors in floating-point arithmetic is not intuitive.

Our work concerns the automatic transformation, at compile-time, of arithmetic expressions in order to improve their numerical accuracy. To transform an expression, we proceed in two phases. In the first phase, we build an under-approximation of all its mathematically equivalent expressions (for a certain definition of mathematical equivalence between expressions). This under-approximation must be representable and tractable in polynomial size while covering as much as possible the exponential number of concrete 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 some expression which minimizes the roundoff errors in the worst case, i.e. for the worst inputs taken in the ranges specified by the user. In other terms, we aim at specializing the source program for ranges of inputs provided 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, summarized in Section 6, requires to explore an exponential space without local selection criterium and we use a limited depth backtracking algorithm and an analysis *à la* Fluctuat [11, 4] to select an accurate expression among all the expressions encountered during the exploration. In this article, we present a new method to generate a large set of arithmetic expressions all mathematically equivalent to an original expression. This kind of semantics-based transformation [3] has been introduced in [10, 12] and the current work strongly improves the existing transformations by permitting the generation of alternative expressions of very different shapes.

Technically, we define a intermediate representation called

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

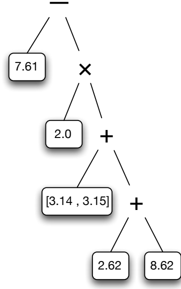

Figure 1: Syntactic tree of expression e.

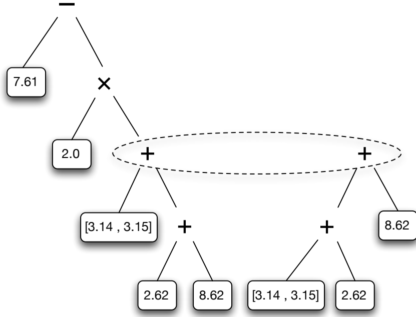

Figure 2: APEG built on e by associativity.

Abstract Program Expression Graph (APEG), inspired from the Equivalence Program Expression Graph (EPEG) structure introduced in [15] for the phase ordering problem in compilers. Our APEGs use abstraction boxes to represent in polynomial size very large sets of mathematically equivalent expressions of different shapes. To prove the correctness of our approach, we introduce a Galois connection between sets of equivalent expressions and APEGs and we introduce an abstract semantics to under-approximate by APEGs the set of evaluation traces of an arithmetic expression up to associativity, commutativity, etc. Finally, we present experimental results obtained with the Sardana tool which implements these techniques.

This article is organized as follows. Section 2 presents an overview of our approach and Section 3 introduces APEGs. Section 4 presents the transformations we apply to APEGs. Section 5 defines the Galois connection between the collecting semantics and APEGs. Section 6 contains a summary of how the profitability analysis works. Finally, Section 7 gives some experimental results.

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. The mathematic equality is defined with respect to a certain set $\triangleright$ of transformation rules of expressions, for example associativity or distributivity. Our goal is to build a tractable abstraction of 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.

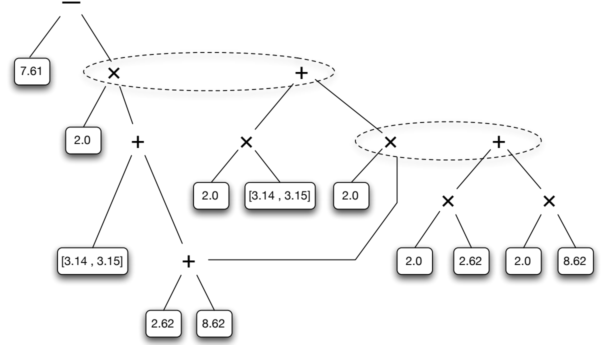

Figure 3: Example of product propagation.

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 $-$), 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.

In the next paragraphs, we detail how an APEG is constructed from the syntactic tree of Figure 1. We attach to each node of the tree a set of equivalent nodes. These sets are denoted in our figures by dashed ellipses. An APEG is always constructed 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 introduce transformations based on associativity, distributivity and commutativity while keeping the size of APEGs polynomial. First, by propagation of 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 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 of the structures: we simply have to add to each multiplication a new branch connected to the lower part of the structure (see Figure 4). Then we

Figure 4: Example of subtraction propagation.

Figure 5: APEG with abstraction boxes.

represent all the possible propagations of the subtraction without growing exponentially.

Finally, we add abstraction boxes into APEGs in the subtrees where an operator is uniformly repeated. 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 expressions contained in the box and for 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 possibly represent an exponential number of expressions. An example of the abstraction boxes we add to the APEG of Figure 2 is given in Figure 5.

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 [15]. 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 propagation or loop unfolding. 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 $\triangleright = \{\triangleright_i,\ 1 \le i \le n\}$, 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 classes of $\triangleright$-equal expressions. Classes make it possible to have in one single data structure many $\triangleright$-equal expressions. Beside classes, we introduce a new type of nodes in APEGs, called abstraction boxes which represent efficiently a possibly very large set of $\triangleright$-equal expressions. An abstraction box is defined by a binary symmetric operator $*$ (like $+$ or $\times$) and a set of operands L. Note that L may contain constants, variables or abstraction boxes (abstraction boxes are a recursive structure). The abstraction box $B = \boxed{*, L}$ represents the set of expressions made of the $*$ operator among 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 recursive box, $\boxed{+,(x_1,x_2,x_3,\boxed{+,(y_1,y_2,y_3)})}$ abstracts all the parsings of $\cup_{x_4 \in Y}\{\sum_{i=1}^{i=4} x_i\}$ where Y denotes all the parsings of $\sum_{i=1}^{i=3} y_i$.
Abstraction boxes are the cornerstone of our abstraction as they make it possible to represent efficiently a huge set of $\triangleright$-equal expressions. Note that, by definition of an abstraction box, an APEG represents only expressions that are $\triangleright$-equal to the original one.

From a formal point of view, the set $\Pi_\triangleright$ of APEGs is defined inductively as the smallest set such that:

(i) $a \in \Pi_\triangleright$ where a is a leaf (a constant or an identifier or an interval $[a, b]$ abstracting all the values x such that $a \le x \le b$),

(ii) $*(lop, rop) \in \Pi_\triangleright$ where $*$ is a binary operator, *lop* and *rop* are APEGs representing the left and right operands of $*$,

(iii) $\boxed{*,(p_1,\ldots,p_n)} \in \Pi_\triangleright$ is an abstraction box defined by the operator $*$ and the APEGs $p_1, \ldots, p_n$ as operands,

(iv) $\langle p_1, \ldots, p_n \rangle \in \Pi_\triangleright$ is a class of $\triangleright$-equal expressions, where $p_1, \ldots, p_n$ are APEGS. We require that $p_1, \ldots, p_n$ cannot be classes of $\triangleright$-equal expressions themselves, i.e. $p_1, \ldots, p_n$ are induced by the cases *(i)* to *(iii)* of the definition.

Nested equivalence classes do not make sense and are prohibited in the last point of the above definition: since these classes contain $\triangleright$-equal expressions, a class $\langle p_1, \ldots, p_n, \langle p'_1, \ldots, p'_m \rangle\rangle$ 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 5. Equivalence classes are represented by dotted 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 an 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 all it only contains the operator $*$.* Partial homogeneity *We say that an APEG is partially homogeneous if it contains one or several sub-APEGs of size at least two which are fully homogeneous.*

We introduce two kinds of transformations. First, we perform the homogenization of the APEG by adding new nodes in order to introduce new homogeneous sub-expressions. Then we apply the expansion functions which insert the abstraction boxes in the homogenized APEGs. All the transformations are designed to be executed in sequence, in polynomial-time.

4.1 Homogenization Transformations

The homogenization transformations insert into an APEG as many $\triangleright$-equal homogeneous expressions as possible.

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 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 illustrate in Figure 6. Note that, as shown in the graphical representation of the transformation given in Figure 6, in both cases (transformation of 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. They are the core of the abstraction process as they represent in a compact way exponentially many $\triangleright$-equal expressions. 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 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 structure, or the leaves of the right part.

For example let us consider the expression $e = \sum_{i=1}^{n} x_i$ and let some index m such that $1 \leq m \leq n$. We can either create a box $B_1 = (\boxed{+, (x_1, \ldots, x_m)})$ or a box $B_2 = (\boxed{+, (x_{m+1}, \ldots, x_n)})$. In the first case we collapse all the parsing of $\sum_{i=1}^{m}$ and keep a certain parsing of $\sum_{i=m+1}^{n}$ (in an englobing expression). In the second case we consider a certain parsing of $\sum_{i=1}^{m}$ plus any parsing of $\sum_{i=m+1}^{n}$, in the abstraction box. Let us call B_1 and B_2 the left and right parts respectively. The horizontal expansion introduces in the class of each binary operator two abstraction boxes: the former containing all the leaves of the left part, and the latter one the leaves of the right part. In addition the horizontal expansion introduces one more abstraction box that contains all the leaves of the whole homogeneous structure, called *global box*. This transformation is illustrated in Figure 7. For an homogeneous part of n leaves we build all the boxes $\boxed{+, (x_1, \ldots, x_m)}$ and $\boxed{+, (x_{m+1}, \ldots, x_n)}$. Hence we generate $O(2n)$ boxes only, 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 that are not contained in this sub-expression. Let us consider an homogeneous structure defined by a set $P = \{p_1, \ldots, p_n\}$ of operands and by a binary operator $*$. Each binary node in the homogeneous structure defines a sub-expression with a set $\{p'_1, \ldots, p'_k\} \subseteq P$ of leaves. The vertical expansion introduces for each binary node contained in the homogeneous structure an abstraction box defined by the $*$ operator and the set $P \setminus \{p'_1, \ldots, p'_k\}$ of leaves. This transformation is illustrated in 7. This transformation introduces $O(n)$ boxes into an ho-

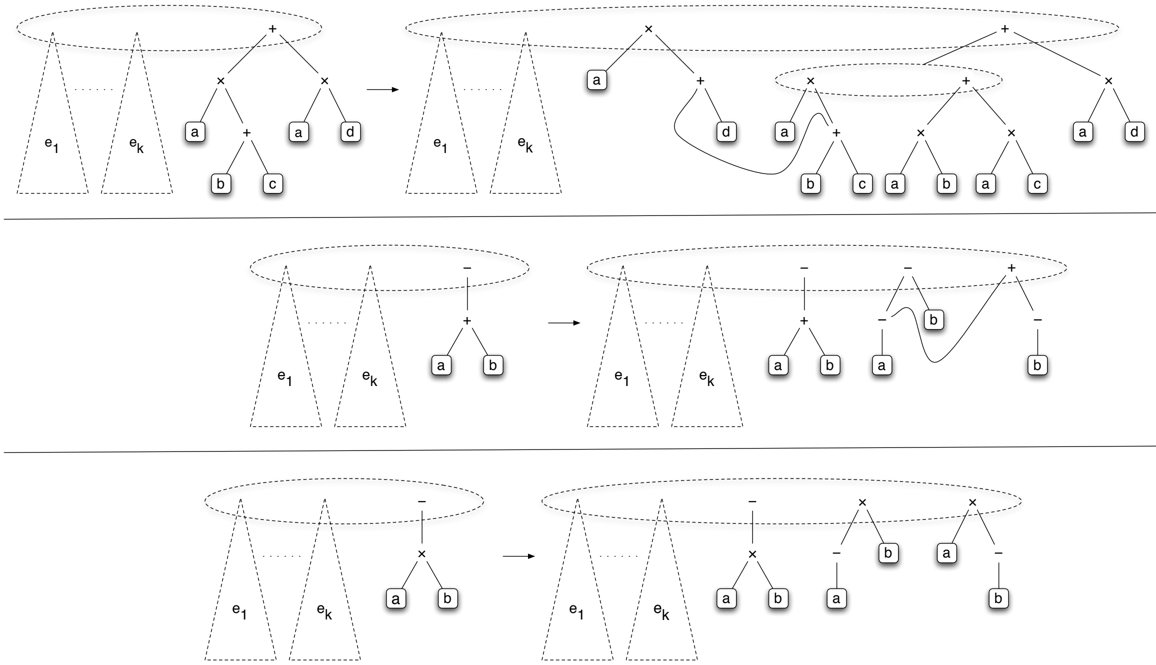

Figure 6: Graphical representation of the homogenization transformations. Dotted nodes represent equivalence classes and $\rhd$-equal expressions $e_1 \dots e_k$ are represented by dashed trees. The top transformation corresponds to the transformation over multiplication and the next two schemes illustrate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

mogeneous part of size n.

4.2.1 Box expansion

The box expansion is designed to add new abstraction boxes to the existing ones. We allow an abstraction box $B' = (\!|\boxed{*', P'}|\!)$ to be part of the operands of an abstraction box $B = (\!|\boxed{*, P}|\!)$, if $* = *'$ then we can merge P and P' into a new abstraction box $B'' = (\!|\boxed{*, P \cup P'}|\!)$. For example, the abstraction box $B = (\!|\boxed{+, (\!|\boxed{+,(a,b,c)}|\!), d, e}|\!)$ yields the abstraction box $B'' = (\!|\boxed{+,(a,b,c,d,e)}|\!)$. Note that the $\rhd$-equal expressions represented by B'' fully includes the $\rhd$-equal expressions represented by B. This transformation is illustrated in 7.

5. CORRECTNESS

In this section, we define the Galois connection which relates sets of traces of the collecting semantics to APEGs.

5.1 Collecting Semantics

Let $(\!|e|\!)_{\rhd}$ be the set of partial traces for the evaluation of e including some steps of transformation of expressions into $\rhd$-equal expressions. In order to define this collecting semantics, we need to introduce some transformation rules of arithmetic expressions into other equivalent expressions. We define $\rhd = \cup_{i=1}^{n} \rhd_i$ with $\forall i,\ 1 \le i \le n,\ \rhd_i \subseteq Expr \times Expr$. We do not require that the $\rhd_i$ relations are transitive since we will compute the transitive closure $\rhd^*$ of $\rhd$. For example, we can set $\rhd_1 = \{((e_1 + e_2) + e_3, e_1 + (e_2 + e_3)) \in Expr^2\ :\ e_1,\ e_2,\ e_3 \in Expr\}$, $\rhd_2 = \{((e_1 + e_2) \times e_3, e_1 \times e_3 + e_2 \times e_3) \in Expr^2\ :\ e_1,\ e_2,\ e_3 \in Expr\}$ and $\rhd_3$ and $\rhd_4$ the relations symmetric to $\rhd_1$ and $\rhd_2$, respectively. Finally, to make $(\!|e|\!)_{\rhd}$ contain the partial traces of evaluation of e, we include in $\rhd$ the relation $\rightarrow$ corresponding to the reduction rules of the standard semantics of expressions. This relation is defined below for addition, subtraction and product. The symbol $*$ stands for $+$, $-$ or $\times$.

$$\frac{v = v_1 * v_2}{v_1 * v_2\ \rightarrow\ v} \qquad \frac{e_1\ \rightarrow\ e_1'}{e_1 * e_2\ \rightarrow\ e_1' * e_2} \qquad \frac{e_2\ \rightarrow\ e_2'}{v_1 * e_2\ \rightarrow\ v_1 * e_2'} \tag{1}$$

We define $(\!|e|\!)_{\rhd}$ as the set $\rhd^*$ of $\rhd$-chains, i.e. the set of all the sequences of form $e \rhd e_1 \rhd \dots \rhd e_n$ for some expressions $e_1, \dots, e_n$ such that $\forall i,\ 1 \le i < n,\ e_i \rhd e_{i+1}$ and $e \rhd e1$.

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 $\Phi\ :\ \Pi_e^{\rhd} \rightarrow \Pi_e^{\rhd}$ 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. $\forall t^\sharp \in \Pi_e^{\rhd},\ t^\sharp \sqsubseteq \Phi(t^\sharp)$, 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

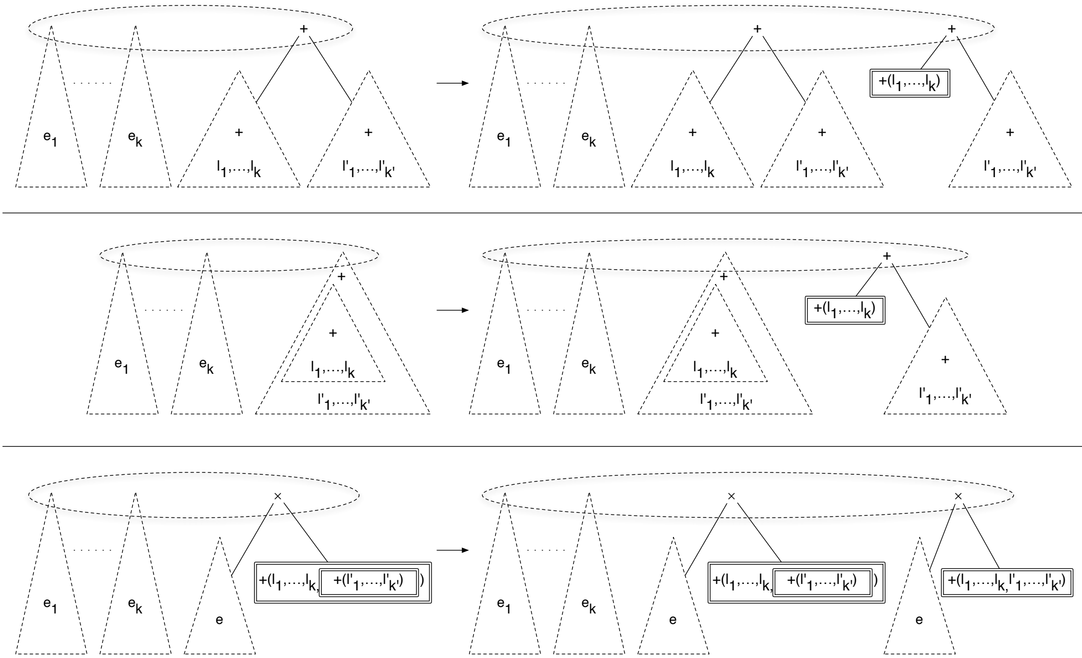

Figure 7: Graphical representation of the expansion transformations. Dotted nodes represent equivalent classes and $\triangleright$-equal expressions $e_1 \dots e_k$ are represented by dashed trees. The dotted triangles with $+(l_1,\dots,l_k)$ written inside represent homogeneous parts and $*$ denotes any binary operator. From top to bottom, the figure represents the horizontal, vertical and box expansion transformations.

$\bot$, $\Phi(\bot)$, $\Phi^{(2)}(\bot)$, ... is increasing. Then C has an upper bound since $\Pi_e^{\triangleright}$ is a CPO. The function Φ being continuous, $\sqcup_{c\in C}\Phi(c) = \Phi(\sqcup C)$ and, finally, by condition c) $\gamma(\llbracket e\rrbracket^\sharp) = \gamma(Fix\ \Phi(\bot)) = \gamma(\sqcup_{c\in C}\Phi(c)) = \gamma(\Phi(\sqcup C)) \sqsubseteq (\!|e|\!)_\triangleright$.

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 $\triangleright$. 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 $\triangleright$ and can be chosen to generate a new expression.

5.2 Abstraction and Concretization Functions

For an initial expression e, the set $(\!|e|\!)_\triangleright$ contains both evaluation steps of the elementary operations and transformations of the expression e into $\triangleright$-equal expressions. The elements of $(\!|e|\!)_\triangleright$ are of the form $e \triangleright e' \triangleright \dots \triangleright e^n$, where $e, e', \dots, e^n$ are $\triangleright$-equal and we may aggregate them into a global APEG since this structure has been introduced to represent multiple $\triangleright$-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 and an index $1 \le i \le n$ such that $p = p_i$,
- 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 e_1 is evaluated first, following t_1, then e_2 is evaluated following t_2 and finally the operation $*$ is performed.

The concretization γ of an APEG $\pi \in \Pi_\triangleright$ 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 $*$, then

$$\gamma(\pi) = \bigcup_{\substack{p_l \in\ \mathcal{C}(lop)\\ p_r \in\ \mathcal{C}(rop)}} \{t_l \circ_* t_r\ : t_l \in \gamma(p_l),\ t_r \in \gamma(p_r)\}, \qquad (2)$$

(iii) if $\pi = \langle p_1,\dots,p_n\rangle$ then $\gamma(\pi) = \bigcup_{i=1}^{n}\gamma(p_i)$,

(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 described in Point *(ii)* of the definition.

5.3 The Abstract Domain of APEGs

In this section, we first 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) $\forall p'_i \in \{p'_1, \ldots, p'_m\}$, where p'_i is not an abstraction box, $\exists p_j \in \{p_1, \ldots, p_n\}$ such that $p_j = p'_i$,

(iii) $\forall p'_i \in \{p'_1, \ldots, p'_m\}$, where 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 as p_i is an abstract box and $p'_i \sqsubseteq_\square p_j$.

We define the join $\sqcup_\square$ of two boxes $B_1 = \boxed{*, (p_1, \ldots, p_n)}$ and $B_2 = \boxed{*', (p'_1, \ldots, p'_m)}$. Let $B_3 = \boxed{*, (p_1, \ldots, p_n, p'_1, \ldots, p'_m)}$. 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 $\pi_1, \pi_2 \in \Pi_\rhd$ we have $\pi_1 \sqsubseteq \pi_2$ if and only if one of the following conditions holds:

(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: $\pi_1 = *(lop, rop)$, where $*$ is a binary operator and lop and rop are its two operands, $\pi_2 = *'(lop', rop')$, $* = *'$, $lop \sqsubseteq lop'$ 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 made of the operator $*$ 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 $\pi_1 = \langle p_1, \ldots, p_n \rangle$, $\pi_2 = *(lop, rop)$, $lop = \{p^l_1, \ldots, p^l_{k_l}\}$, $rop = \{p^r_1, \ldots, p^r_{k_r}\}$ and $\forall p_i \in \pi_1, \exists p^l_j \in lop$ and $\exists p^r_k \in rop$ such that $p_i \sqsubseteq *(p^l_j, p^r_k)$.

In order to define $\pi_1 \sqcup \pi_2$, with $\pi_1, \pi_2 \in \Pi_\rhd$, we observe first that π_1 and π_2 aim at containing $\rhd$-equal expressions only. 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 $\perp$ as the empty APEG, and $\top$ as the APEG built with all the possible expression transformations of $\rhd$. We have the following Galois connection between the collecting semantics and the APEGs:

$$\langle \wp((\!|e|\!)_\rhd), \subseteq \rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle \Pi_e^\rhd, \sqsubseteq \rangle \tag{3}$$

where $\wp(X)$ denotes the powerset of X.

6. PROFITABILITY ANALYSIS

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

We use an arithmetic which computes both the machine result of some sequence of operations and the numerical error arising during this computation. This arithmetic corresponds to special cases of the semantics of floating-point error with errors introduced in [9, 12]. We consider an arithmetic in which error terms are attached to the floating-point or fixed-point numbers. 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 overapproximation 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 [16]. 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. For example, in our arithmetic, using the IEEE754 double precision, the result of the operation $1 \div 3$ is $3.333333333333333 \cdot 10^{-1}$ with an interval of error of $[1.850371707708594 \cdot 10^{-17}, 1.850371707708595 \cdot 10^{-17}]$.

In our arithmetic, a pair (f, e) made of a floating or fixed-point number f and of an error e is used to represent the real number $r = f + e$ rounded to f in the current computer format. In practice, this is extended to intervals. By definition, the pair of intervals $([\underline{f}, \overline{f}], [\underline{e}, \overline{e}])$ represents the set of floating or fixed-point numbers belonging to $[\underline{f}, \overline{f}]$ whose error with respect to the real number they aim at representing belongs to $[\underline{e}, \overline{e}]$:

$$\begin{array}{c} ([\underline{f}, \overline{f}], [\underline{e}, \overline{e}]) \\ = \\ \{(f, e) \in \mathcal{F} \times \mathcal{R} \ : \ f \in [\underline{f}, \overline{f}], \ e \in [\underline{e}, \overline{e}]\} \end{array} \tag{4}$$

In Equation (4), $\mathcal{F}$ denotes the current format used to represent the number in memory (single or double precision) and $\mathcal{R}$ denotes the set of real numbers.

Error terms are propagated among computations as defined in equations (5-7). The error on the result of some operation $x \diamond y$ is the propagation of the errors on x and y through the operator $\diamond$ plus the new error due to the rounding of the result of the operation itself.

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

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

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

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. The elementary operations in our arithmetic are defined in equations (5-7).

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 arithmetic 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 $p_2 = \langle p''_1, \dots p''_n \rangle$. 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. For example, we may have $p'_1 = *(q_1, q_2)$ with $q_1 = \langle q'_1, \dots q'_r \rangle$ and $q_2 = \langle q''_1, \dots q''_s \rangle$ and we should consider all the expressions $*(*(q_u, q_v), p''_j)$ for $1 \leq i \leq n$, $1 \leq j \leq m$, $1 \leq u \leq r$ and $1 \leq v \leq s$. To cope with this combinatorial explosion, we only 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, this consists of considering only the best way to evaluate p'_1 when examinating $*(p_1, p_2)$.

For a box $B = \boxed{*, (p_1, \dots, p_n)}$, we use an heuristic which generates an accurate expression (yet not always optimal). The operands are sorted by increasing absolute values and are then composed. For example, for the box defined by $B = \boxed{+, ([0, 2.0], [-5, 0], [-0.5, 0.5])}$ we will generate the expression $([-0.5, 0.5] + [0, 2.0]) + [-5, 0]$.

7. EXPERIMENTAL RESULTS

In this section, we present experimental results obtained using our tool, Sardana. First, in Section 7.1, we aim at showing on specific examples that our techniques makes it possible to generate expressions whose shape strongly differs from the original ones and which improve the numerical accuracy. Next, in Section 7.2, we aim at showing the generality of our techniques by presenting statistical results on large sets of randomly generated expressions.

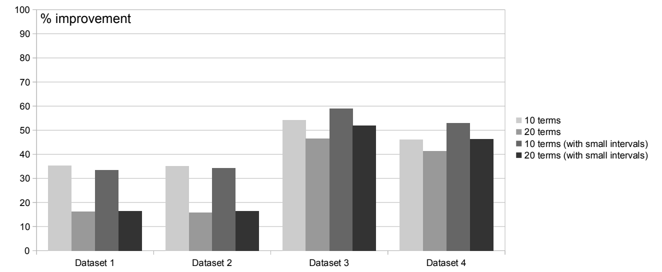

Figure 8: Average improvement for summations.

7.1 Case studies

Figure 7.1 shows our results for our chosen examples. The first line of the row describes the input of the program and the original error while the second line shows the expression which has been generated as well as the new error bound. All these results have been obtained in less than one second on a simple laptop computer (while the number of concrete traces is huge even for small expressions).

First, let us consider a simple example: the expression $x^2 + x$, with x defined by an interval $x \in [800.0, 1000.0]$. All the computations are carried out in IEEE754 double precision. With $x \in [800.0, 1000.0]$, a factorized form of the expression is better because $x^2 \gg x$. We can see in Figure 7.1 that we successfully generated the expression $x \times (x + 1)$ instead of $x^2 + x$. In the second example, we complicate a bit the former example. We replace x by a sum $a_1 + (a_2 + (a_3 + a_4))$ where $a_i \in [200.0, 250.0]$, $1 \leq i \leq 4$. We intend to have the same result than earlier, disregarding of the shape of the sums below, as the sums themselves do not introduce significative rounding errors. We obtain:

$$S = ((((a_4 + a_3) + a_1) + a_2) \times ((((a_4 + a_3) + a_1) + a_2) + 1))$$

The horizontal expansion has efficiently collapsed each sum $\sum_{i=1}^{4} a_i$ into one abstract box, and the analysis has focused on the product. This has drastically narrowed down the search space. It also explains why, in our results, the terms of the sum are not in the same order than earlier. It also points out that our approach with abstraction boxes is efficient on all the subparts of abstract APEGs.

To illustrate how the vertical expansion works, we consider an unbalanced summation (Example 3) where X is far greater than a, b, c, d. In the initial expression, X is added right at the beginning of the summation and tends to absorb the other values [5]. A better way to perform this summation is to add X at the very end, allowing the other values to be accumulated in a result that X cannot absorb. In this example, the covering function φ_2 inserts into the APEG the node $+(X, \boxed{+, (a, b, c, d)})$, which represents exactly the expressions where X is added at last, and where the parsing of the sum of a, b, c, d is not relevant. Our analysis returns the expression $(((a + b) + c) + d) + X$ (in this example the abstraction box has generated by chance a sum where the terms a, b, c and d are in the same order as before).

Example 4 shows that our covering functions also have limitations. If we modify the former example and insert X twice, at two different places in the sum, our analysis is not able to

$n°$	expression	value bound	errors bounds
1	$x^2 + x$ $x \times (x+1)$	$x \in [800, 1000]$	$[-0.0625, 0.0625]$ $[-0.0617, -0.0617]$
2	$(\sum_{i=1}^{4} a_i)^2 + (\sum_{i=1}^{4} a_i)$ S	$a_i \in [200, 250]$	$[-0.215, 0.215]$ $[-0.153, -0.153]$
3	$a + (b + (c + (d + X)))$ $(((a + b) + c) + d) + X$	$a, b, c, d \in [0.1, 0.2],\ X \in [100.0, 101.0]$	$[-1.52e^{-5}, 1.52e^{-5}]$ $[-0.38e^{-5}, 0.38e^{-5}]$
4	$(a + X + b) + (a + X + c)$ $((c + a) + X) + ((a + b) + X)$	$a, b, c \in [0.1, 0.2],\ X \in [100.0, 101.0]$	$[-2.28e^{-5}, 2.28e^{-5}]$ $[-1.52e^{-5}, 1.52e^{-5}]$
5	$P_1(x)$ $P_1^t(x)$	$a_i \in [50.0, 200.0],\ x \in [100.0, 101.0]$	$[-1.01e^{-8}, 1.01e^{-8}]$ $[-0.82e^{-8}, 0.82e^{-8}]$
6	$P_2(x)$ $P_2^t(x)$	$a_i \in [50.0, 200.0],\ x \in [100.0, 101.0]$	$[-5.93e^{-7}, 5.93e^{-7}]$ $[-5.89e^{-7}, 5.89e^{-7}]$

Figure 9: Experimental results obtained with the Sardana tool.

extract both occurrences of X out of the sum. Nevertheless it is able to locally modify the inner sums in order to put the occurrences of X at the best place in the sub-expressons.

We end this section with two additional larger examples. In examples 5 and 6, we consider two polynomial functions of degree 6:

$$P_1(x) = a_0 + (a_1x + (a_2x^2 + (a_3x^3 + (a_4x^4 + (a_5x^5 + a_6x^6)))))$$

$$P_2(x) = (((((a_0 + a_1x) + a_2x^2) + a_3x^3) + a_4x^4) + a_5x^5) + a_6x^6$$

These functions represent the same polynomial but with opposite parenthesisations. Let P_1^t and P_2^t be the results of our transformation:

$$P_1^t(x) = (a_0 + a_1x) + ((((((a_5x) + a_4) + a_6x^2) \times x^4) + (a_3x^3)) + a_2x^2)$$

$$P_2^t(x) = (a_6x^6 + (a_5x^5 + (a_4x^4 + (a_3x^3 + ((a_0 + a_1x) + (a_2x^2))))))$$

P_1 is less accurate than P_2 as the order of the summation is from the lowest degree to the highest, which favours the absorption of intermediate results. It is not easy to explain precisely the shape of P_1^t and P_2^t, but it is obvious that in order to reduce the error induces by P_1, our analysis has to generate an expression whose shape is very different from the original one. Conversely, P_2 and P_2^t are rather similar as P_2 introduces less rounding error. let us remark that this strongly depends on the value of x and on the values of the coefficients a_i. We may claim that these expressions would never have been produced by hand by a programmer.

7.2 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). We use datasets inspired from [7] which are designed to illustrate the different 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 datasets:

- Dataset 1: Only positive sign, 20% of large values among small values. There are absorptions and accurate algorithms should first sum the smallest terms,
- Dataset 2: Only positive sign, 20% of large values among small and medium values. The best results should be obtained with sums in increasing order,
- Dataset 3: Both signs, 20% of large values that cancel, among small values. The most accurate algorithms should sum the largest values first. In a more general case, the best algorithms should sum in decreasing order of absolute values. It is a classic ill-conditioned summation,
- Dataset 4: Both signs, few small values and same proportion of large and medium values. Large and medium values are ill-conditioned. The best algorithms should sum in decreasing order of absolute values.

For each of these datasets, we present in Figure 8 the average improvement on the error bound, i.e. the percentage of reduction of the error bound. We test each dataset on different configurations: with 10 or 20 terms, and with several widths of intervals: small width (interval width about 10^{-12} times the values) or large width (interval width about 10 percent of the values). Each result is an average of the error reduction on 1000 randomly generated initial parsings of the sums. For these datasets, 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.

Figures 10 and 11 present the average improvement on the roundoff error bound for more complex expressions. We use the same datasets as before, however we use more complex expressions some additions by products and subtractions. Figure 10 presents the results obtained for randomly generated expressions with 45% of sums, 10% of products and 45% of subtractions. The Figure 11 presents the results obtained for expressions containing 50% of additions, 25% of products and subtractions. As shown in these figures, the accuracy of the evalaution is improved of 20% in average. This states that our tools transforms the original expressions into new expressions whose evalution yields a roundoff error 20% smaller than the original one in the worst case, for any concrete dataset taken into the intervals for which the specialization has been performed.

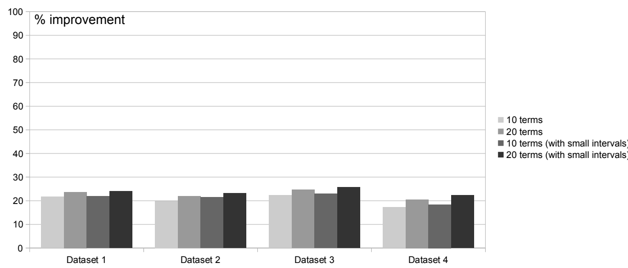

Figure 10: Average improvement for complex expressions with 45% of plus, 10% of multiplications and 45% of subtractions.

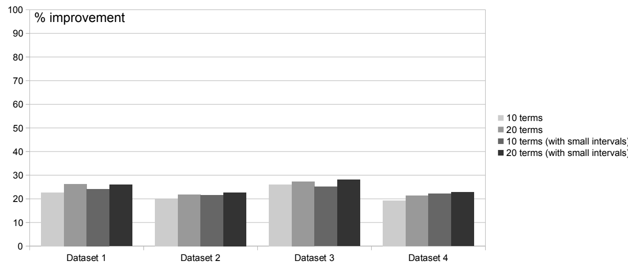

Figure 11: Average improvement for complex expressions with 50% of plus, 25% of multiplications and 25% of subtractions.

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 regrouping 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.

10. 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'77*, pages 238–252. ACM Press, 1977.

[3] P. Cousot and R. Cousot. Systematic design of program transformation frameworks by abstract interpretation. In *POPL'02*, pages 178–190. ACM Press, New York, NY, 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'09*, 2009.

[5] D. Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 23(1), 1991.

[6] E. Goubault, M. Martel, and S. Putot. Asserting the precision of floating-point computations: a simple abstract interpreter. In *ESOP'02*, number 2305 in LNCS, pages 209–212, 2002.

[7] Philippe Langlois, Matthieu Martel, and Laurent Thévenoux. Accuracy Versus Time: A Case Study with Summation Algorithms. In *PASCO '10*, pages 121–130. ACM, 2010.

[8] M. Martel. An overview of semantics for the validation of numerical programs. In *VMCAI'05*, number 3385 in LNCS, pages 59–77. Springer-Verlag, 2005.

[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'07*, number 4634 in LNCS. Springer-Verlag, 2007.

[11] M. Martel. Program transformation for numerical precision. In *PEPM'09*, pages 101–110, 2009.

[12] Matthieu Martel. Enhancing the implementation of mathematical formulas for fixed-point and floating-point arithmetics. *Journal of Formal Methods in System Design*, 35, 2009.

[13] D. Monniaux. The pitfalls of verifying floating-point computations. *TOPLAS*, 30(3), May 2008.

[14] R. E. Moore, R. B. Kearfott, and M. J. Cloud. *Introduction To Interval Analysis*. Cambridge Uni Press (CUP), 2009.

[15] R. Tate, M. Stepp, Z. Tatlock, and S. Lerner. Equality saturation: A new approach to optimization. In *POPL '09*, pages 264–276. ACM, 2009.

[16] Torbjorn Granlund and the GMP development team. *The GNU Multiple Precision Arithmetic Library*, 5.0.2 edition, 2011. `http://gmplib.org`.