Synthesizing Accurate Floating-Point Formulas
Arnault Ioualalen, Matthieu Martel

To cite this version:
Arnault Ioualalen, Matthieu Martel. Synthesizing Accurate Floating-Point Formulas. ASAP: Application-Specific Systems, Architectures and Processors, Jun 2013, Washington, DC, United States. pp.113-116, 10.1109/ASAP.2013.6567563. hal-00835736

HAL Id: hal-00835736
https://hal.archives-ouvertes.fr/hal-00835736
Submitted on 21 Jun 2013

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
Synthesizing Accurate Floating-Point Formulas†

Arnault Ioualalen1,2 and Matthieu Martel1,2

1Univ. Perpignan Via Domitia, Digits, Architectures et Logiciels Informatiques, F-66860, Perpignan, France
2Univ. Montpellier II, CNRS, LIRMM, UMR 5506, F-34095, Montpellier, France
Email: {arnault.ioualalen, matthieu.martel}@univ-perp.fr

Abstract—Many critical embedded systems perform floating-point computations yet their accuracy is difficult to assert and strongly depends on how formulas are written in programs. In this article, we focus on the synthesis of accurate formulas mathematically equal to the original formulas occurring in source codes. In general, an expression may be rewritten in many ways. To avoid any combinatorial explosion, called APEG, enabling us to represent many equivalent expressions in the same structure. In this article, we specifically address the problem of selecting an accurate formula among all the expressions of an APEG. To validate our approach, we present experimental results showing how APEGs, combined with profitability analysis, make it possible to significantly improve the accuracy of floating-point computations.

I. INTRODUCTION

Most critical control systems of recent planes, spatial vehicles or power-plants rely on floating-point computations [1] yet this arithmetic is not intuitive [11]. Indeed, it is very difficult to predict by hand the accuracy of the evaluation of a formula, given certain ranges for the inputs and, recently, static analysis techniques have been developed to infer safe error bounds for the computations arising in critical programs written in C [6], [7]. However these techniques do not indicate how to improve the accuracy if the inferred error bounds are not satisfying.

Our work concerns the synthesis at compile-time of accurate formulas, for given input ranges, to replace the expressions written by the programmers in source codes [10]. 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 [5]. We build Abstract Program Equivalence Graphs (APEGs) to represent in polynomial size an exponential number of mathematically equivalent expressions [8]. APEGs are abstractions of the Equivalence Program Expression Graphs introduced in [14]. The concretization of an APEG yields expressions of very different shapes and accuracies. To synthesize expressions from APEGs, we use a profitability analysis which searches the most accurate expressions among all the expressions represented by the APEG.

This article focuses on our profitability analysis. An APEG is an abstraction of an exponential number of expressions and the profitability has to extract an accurate formula. We compute safe error bounds using established static analysis techniques for numerical accuracy [10] and we use a limited depth search algorithm to explore the APEG structure. In addition, APEGs contain abstraction boxes representing any parsing of a sequence of operations defined by a set of operands and one commutative operator. We also define a way to synthesize an accurate formula from abstraction boxes.

For tractability reasons, we require our profitability analysis to be polynomial in the size of the APEGs. The APEGs representing an exponential number of expressions, our profitability is then a heuristic and we present experimental results to assert its efficiency. We implemented our techniques in a tool, named Sardana which takes as entry Lustre programs [4]. Inputs are represented by abstract streams which indicate, at each instant, a range for the values of the variables and a range for the roundoff errors on these variables. We present experimental results showing how our techniques improve the numerical formulas arising of several relevant pieces of codes extracted from an industrial critical software coming from aeronautics.

II. APEG CONSTRUCTION

In this section, we present our intermediate representation of programs, called Abstract Program Expansion Graph (APEG). This intermediate representation is inspired from the EPEGs introduced in [14]. However, as an EPEG is not necessarily complete, we define the APEGs as an intermediate structure between the initial PEGs and the theoretical complete EPEGs which can be intractable or infinite. The main objective of APEGs is to use abstractions in order to remain polynomial in size while still representing the largest number of equivalent expressions. APEGs contain a compact representation of many transformations of expressions in abstraction boxes which allow one to represent very large sets of expressions in polynomial size, despite that these expressions are of very different shape. An abstraction box is defined by a commutative binary operator, such as + or ×, and by a list of nodes which correspond to the operands. These nodes can be either constants, variable identifiers, sub-trees, equivalence classes or abstraction boxes. An abstraction box stands for all the parsings of the given leaves using the binary operator. For example, +, (a, b, c, d) stands for all parsings of the sum a + b + c + d. Also, ×, (x1, x2, x3) stands for all the summations of the sum a + b + c + X, where X stands for any parsings of the product x1 × x2 × x3. So, an abstraction
III. Profitability Analysis

To compute safe bounds on the numerical accuracy of arithmetic expressions, we use abstract values $(x^\sharp, \mu^\sharp) \in \mathbb{E}^2$ where $x^\sharp$ and $\mu^\sharp$ are intervals whose bounds are floating-point numbers, and where $x^\sharp$ represents the interval of values of the input and $\mu^\sharp$ represents the intervals of errors on the input $[10]$. A value $(x^\sharp, \mu^\sharp)$ abstracts a set of concrete values $\{x_i, \mu_i, \ i \in I\}$ by intervals in a component-wise way.

For an arithmetic expression, the propagation of roundoff errors corresponds to the semantics of [10] and is given in Figure 2. The abstract function $\uparrow o$ corresponds to the concrete function $\downarrow o$ which calculate the difference between a real value and its rounded floating-point value. We have:

$$\downarrow o ([x, \mu]) = [\uparrow -\infty (x), \uparrow +\infty (\mu)]$$

The function $\downarrow o^2$ is a safe abstraction of $\downarrow o$, i.e. $\forall x \in [x, \mu], \downarrow o (x) \in \downarrow o^2 ([x, \mu])$. For example, if the current rounding mode $o$ is to the nearest, one may choose $\downarrow o^2 ([x, \mu]) = [-y, y]$ with $y = \frac{1}{2} \mathrm{ulp}(\max(|x|, |\mu|))$ (2)

where the unit in the last place $\mathrm{ulp}(x)$ is the weight of the least significant digit of the floating-point number $x$ [13]. 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. Finally, the semantics of the multiplication comes from the development of $(x^\sharp_1 + \mu^\sharp_1) \times (x^\sharp_2 + \mu^\sharp_2)$. The semantics of other operations is described in [10]. We use the former arithmetic for the elementary operations between floating-point values. In Lustre, values are streams and the operations have to be extended to streams.

As a synchronous language, Lustre [4] handles streams of values recording the values of a program point at each time instant. For example, let us consider the following program:

$$a = 1.0 / 3.0; \quad b = 1.0 - a \ast \pre(b);$$

A constant stream mapping each instant to the internal representation of $\frac{1}{3}$ is associated to $a$. The stream $b$ is the stream whose value is $b(0) = 1.0$ at time 0 and whose value at time $i$ is $b(i) = a(i) \ast b(i - 1)$.

Following the notations of [3], we denote $s = \langle m, t_0 : x_0 \land t_1 : x_1 \land \ldots \land t_n : x_n \rangle$ the stream $s$ such that, at each instant $i$ mod $m$, $s(i) = x_k$ if $t_k \leq i$ mod $m < t_{k+1}$ for some $k \in [0, N - 1]$ or $s(i) = x_N$ if $t_N \leq i$ mod $m < m$. By extension, we consider that if $m = 0$ then no modulo holds and $\forall i \geq t_n, s(i) = x_N$. We also assume that always $t_0 = 0$.

Our streams associate at each instant a value in the domain $\mathbb{E}$ of intervals of floating-point numbers with errors introduced in Section III. Hence, a stream $s \in \mathcal{S}$ is a mapping $s : \mathbb{N} \rightarrow \mathbb{E}$ where $\mathbb{N}$ denotes the set of non-negative integers. Coming back to our former example, the stream associated to $a$ is

$$0, 0 : \{(3.33333333333333333E-1, 3.33333333333333E-1), \langle 1.85037170770855E-17, 1.85037170770861E-17 \rangle \}$$

and the stream associated to $b$ is

$$4, 0 : \{(2.09075158128704E-7, 1.0), \langle -6.21498708142+224E-23, 3.39955157246792E-22 \rangle \} \land 1 : \{(2.09075158128704E-7, 3.39955157246792E-22), \langle -2.77556237867805E-17, 4.62296567806975E-17 \rangle \} \land 2 : \{(2.09075158128704E-7, 1.11111111111112E-17), \langle -1.6100768541812E-17, 2.8526996899166E-17 \rangle \} \land 3 : \{(2.09075158128704E-7, 3.73070707076372E-2), \langle -8.86090704903589E-18, 1.50484528166525E-17 \rangle \}$$

Elementary operations are applied at each instant, i.e if $s = \langle m, 0 : x_0 \land t_1 : x_1 \land \ldots \land t_n : x_n \rangle$ and $s' = \langle m', 0 : x'_1 \land t'_1 : x'_1 \land \ldots \land t'_n : x'_n \rangle$. The notation $s'^\# (m', 0 : x'_1 \land t'_1 \land \ldots \land t'_n : x'_n)$ is used when the expression is composed of arithmetic expressions, we use abstract values $\langle x^\sharp, \mu^\sharp \rangle$.
mizes the roundoff errors. As our values are element of \( E \),
we have to define in what sense we aim at minimizing the
errors. First, let us introduce some notations. For a value
\( v = (x^4, \mu^2) \) \( \in E \), let \( \mathcal{E}(v) = \mu^2 \). The function \( \mathcal{E} \) gives
the error term of an abstract value. Next, if \( \mathcal{E}(v) = \mu^2 = [\mu, \overline{\mu}] \),
then \( M_E^x(v) = \overline{\mu}, M_E^\mu(v) = \mu, \) and \( M_E(v) = \max(|\mu|, |\overline{\mu}|) \).
In other words, \( M_E^x(v), M_E^\mu(v) \) and \( M_E(v) \) denote the upper
and lower bounds of the error and the maximal absolute error
bound, respectively. We consider several orders:

- **Strict order:** \( s \prec_s s' \) if \( \forall i \in \mathbb{N}, \mathcal{E}(s(i)) \subseteq \mathcal{E}(s'(i)) \).
  This order requires that, if \( s \prec_s s' \) then at any time
  the error bound on \( s \) is less than the error bound on \( s' \).
  The accuracy is improved at each instant.

- **Max order:** \( s \prec_m s' \) if \( \max_{i \in \mathbb{N}} M_E(s(i)) \leq M_E(s'(i)) \).
  This order only considers the worst errors, i.e., \( s \prec_m s' \)
  if the worst intensity of the error possibly associated to
  a value at a given instant is always smaller in \( s \) than in \( s' \).
  Elsewhere, the errors may be greater in \( s' \) than in \( s \).

- **Integral order:** \( s \prec_i s' \) if \( \sum_{i=0}^{m-1} M_E^*(s(i)) - M_E^*(s'(i)) \leq \sum_{i=0}^{m-1} M_E^*(s'(i)) - M_E^*(s(i)) \).
  This order compares the integrals of the errors \( \mathcal{E}(s) \) and \( \mathcal{E}(s') \).
  We have \( s \prec_i s' \) if the sums of the errors at
  each instant is smaller in \( s \) than in \( s' \). If \( s \prec_i s' \) then
  at some instant \( i \), the error \( s(i) \) may be greater than the
  error \( s'(i) \). If \( m = 0 \) or \( m' = 0 \), the integral is computed
  up to \( \max(m, m') - 1 \). If \( m = 0 \) and \( m' = 0 \), the integral
  is computed up to \( \max(t_N, t'_N) \).

The Strict order \( \prec_s \) would require to synthesize a program
which always improves the error bounds on the computed values.
We consider that this order is too restrictive and we do
not use it in practice. The Max order \( \prec_m \) may be
in certain applicative contexts where the main objective is to
lower the worst error bound and where the average error is
not relevant. The Integral order \( \prec_i \) gives a measure of the
average error. We consider this order as the most interesting.
In practice, our experiments confirm that the integral order is
the order for which we may optimize the most the programs.

To synthesize an optimized program, the profitability has
to be performed on the APEGs as defined in Section II.
A main difficulty is that, thanks to equivalence classes, an
APEG may represent an exponential number of expressions
whose accuracy should be individually evaluated. To cope with
this combinatorial explosion, we use a limited depth search
strategy with memoization. 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.

Algorithm 1 illustrates how we perform the profitability
when the depth is set to 1. The operator \( \vdash \) appends a value to a
list. It considers, for each node in the given equivalence class,
the cartesian product of the elements of the node. It uses the
equivalence classes of the node parameters and evaluates the
roundoff error generated by the operator using the semantics
introduced above. Then it returns the minimal roundoff error
for the desired order (\( \prec_s \), \( \prec_m \) or \( \prec_i \)) after memoization of the
accuracies of the expressions encountered. Algorithm 1 does
not detail how to synthesize the final expression once we have
found the minimal stream. This step is a simple propagation of
the expressions along with the streams.

The next point concerns the synthesis of an expression for
an abstraction box \( B = \{ \ast, (p_1, \ldots, p_n) \} \). In this case, we use an
heuristic which generates an accurate expression (yet not
always optimal). This heuristic is a greedy algorithm which
searches at each step the pair \( p_i \) and \( p_j \) such that \( \sharp^\ast(p_i \ast p_j) \)
is minimal. Once it finds \( p_i \) and \( p_j \) it replaces both terms in the
box by a new term \( p_{ij} \) whose accuracy is equal to \( \sharp^\ast(p_i \ast p_j) \).
This heuristic computes in \( O(n^3) \) iterations.

### IV. Case Study: Avionic Benchmarks

We present in this section several experimental results
obtained on pieces of code extracted from embedded critical
industrial avionic codes. These experimental results concern
in one hand programs using the IEEE-754 binary 32 format
(Table I), and, on the other hand, programs using the IEEE-754
binary 64 format (Table II). For both floating-point formats,
we have tested each program for many contexts, each having its
own input values. Again, input values are described by means
of streams of intervals of floating-point numbers. All the
contexts we used have been kindly provided by the ASTRÉE
team [2] who currently analyzes these programs and have
access to realistic simulations of them. Note that even if some
programs appear in both tables these programs have different
codes and performances, yet they aim at achieving a similar
task. In this case study, we use the Integral order. To perform
our benchmarks, we used for each program and each context
three identical scenarios in which Sardana computes streams of
size 5, 15 or 35 instants. For all the contexts of each program,
we present in Tables I and II the improvement of the integral value of the errors for these scenarios. For industrial property reasons, we denote the programs used by the following generic terms: Interpol stands for a first order interpolation, L-P filter are respectively a first order low-pass filter, H-P filter is a first order high-pass filter, Transfer denotes a second order transfer function between two inputs, and Sqrt is a polynomial interpolation of the square root. We can draw two conclusions with these experiments. First, in many cases, the gain in accuracy decreases as the length of the streams increases. This can be observed on all low-pass filters described here. We explain this phenomenon by the fact that in most of these cases the values generated by the program tend towards zero and the errors on these values then decreases as well over time. Thus, the gain at each step of the integral tends to become smaller leading the integral gain to decrease as well. This conclusion is even more accurate to us, that in the other cases where the values are growing without limit (like with the high-pass filter in Table I or the Transfer program in Table II) the gain on the integral value increases as the stream length increases. The second conclusion is that our approach is able to improve the overall numerical accuracy of very different programs in many different contexts. For most programs this gain is between 2% and 20% for 50% to 90% of the contexts. For the few programs we are unable to improve such as Interpol in Table I, we may argue that these programs do not allow much syntactic transformations.

V. CONCLUSION

In this article we have presented a new profitability heuristic which allows us to extract a more accurate version of a program from our intermediate representation called APEG. This article briefly describes how APEGs are constructed, and how they are able to represent many equivalent versions of a program in order to find one with better numerical accuracy. We have designed a profitability analysis which runs in polynomial time recursively into the APEG and synthesizes a well-formed, new, but yet mathematically equivalent, version of a program. Our experimental results show significant improvement for real case examples of industrial avionic code. We believe that our approach could be extended in many ways, we are confident that the profitability heuristic we currently use could be improve in order to synthesize even more accurate programs. Finally we consider that many other orders over the streams of values could be defined, and may improve the quality of the profitability heuristic we use.

ACKNOWLEDGMENT

We thanks Alexandre Chapoutot for his advises, Antoine Miné and all the ASTRÉE team for their contribution on the avionic benchmarks, also Laurent Thévenoux and Christophe Mouilleron for generating the evaluation schemes of expressions.

REFERENCES