Computer Science • 21(2) 2020 https://doi.org/10.7494/csci.2020.21.2.3421

M.G. Thushara
K. Somasundaram

FORWARD AND BACKWARD STATIC ANALYSIS FOR CRITICAL NUMERICAL ACCURACY IN FLOATING-POINT PROGRAMS

Abstract *In this article, we introduce a new static analysis for numerical accuracy. We address the problem of determining the minimal accuracy on the inputs and on the intermediary results of a program containing floating-point computations in order to ensure the desired accuracy of the outputs. The main approach is to combine a forward and backward static analysis, done by abstract interpretation. The backward analysis computes the minimal accuracy needed for the inputs and intermediary results of the program in order to ensure the desired accuracy of the results (as specified by the user). In practice, the information collected by our analysis may help optimize the formats used to represent the values stored in the variables of the program or to select the appropriate sensors. To illustrate our analysis, we have shown a prototype example with experimental results.*

Keywords abstract interpretation, backward static analysis, floating-point numbers, round-off errors, abstract domain

Citation Computer Science 21(2) 2020: 179–192

Copyright © 2020 Author(s). This is an open access publication, which can be used, distributed and reproduced in any medium according to the Creative Commons CC-BY 4.0 License.

1. Introduction

Numerical accuracy is a critical point in safe computations when it comes to floating-point programs. Given a certain accuracy for the inputs of a program, the static analysis computes a safe approximation of the accuracy on the outputs. This accuracy depends on the propagation of the errors on the data and on the round-off errors on the arithmetic operations performed during the execution.

In programs with floating-point computations, it is demanding to have numerical accuracy in the results. Our approach is to combine a forward and a backward static analysis, done by abstract interpretation. The forward analysis is a classical approach where the errors on the inputs and on the results of the intermediary operations are safely propagated to determine the accuracy of the results. Based on the results of the forward analysis and on assertions indicating the accuracy required by the user for the outputs at the end of the execution, the backward analysis will be carried out. Backward analysis computes the minimal accuracy needed for the inputs and intermediary results of the program in order to satisfy the assertions made. In order to refine the results until a fixed-point is reached, the forward analyses and backward analyses can be applied repeatedly.

Such static analysis are useful in several safety critical contexts. Here we use our analysis as a general case which can be applied in many of the safety critical applications. For instance, the explosion of the rocket-Ariane 5 [1], owing to a software error in the inertial reference system. Specifically, a 64-bit floating-point number was converted to a 16 bit signed integer which was larger than 32,767, the largest integer in a 16 bit signed integer, and that lead to the failure. Another instance was Patriot Missile [17] failed in detecting and intercepting an incoming Iraqi Scud missile and killing 18 American army men during the Gulf war. The cause of the incident was an inaccurate calculation of the time due to computer arithmetic errors.

Technically, we use abstract values written $[a, b]_p$ where a and b are floating-point numbers defining an interval and p is an integer giving the accuracy. Intuitively, $[a, b]_p$ is the set of numbers between a and b which have at least p correct digits.

2. Background

Abstract interpretation(AI) [7] is a theory of approximation in the field of semantics-based program data flow analysis. The program execution is imitated by AI by abstracting the possibles paths/states and then execute the abstraction of the program. These abstractions are referred as abstract domain. To reach a fixed point, AI uses ordering on abstract values. Top value represents all possible values and is denoted as $\top$. Some times it may take many iterations to reach a fixed point, for speeding up this process AI uses widening operation [4] using the widening operator denoted as $\bigtriangledown$.

Interval is considered the basic integer abstract domain. Figure 1 shows the mapping of integers to the interval abstract domain. The abstract interpretation of

of an integer expression is negative, positive or zero. The abstract domain which is interval is sign interval. Interval domains represents the minimum and the maximum number of closed interval. Interval domain is useful in many cases where the program is usually bounded by some minimum and maximum. Consider a loop, the iterated variable is initialized to minimum or maximum value at the entry of the loop and until a maximum or minimum value is reached it is incremented or decremented.

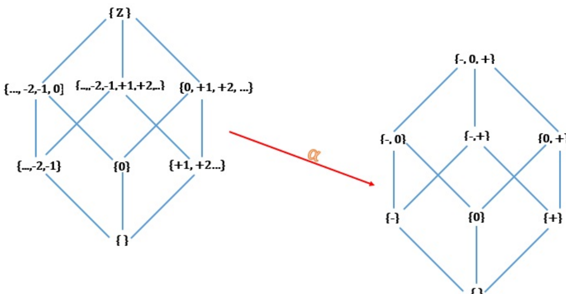

Figure 1. Abstract interpretation: a lattice model for integers

There are many eminent scientists who work in this area. Abstract analysis on floating-point computations lead to precision loss [14]. Goubault in this paper points out that a wrong estimate of precision can be very expensive in safety critical applications. IEEE standard [15] gives the specifications for floating-point numbers. It also gives a guidelines for floating-point arithmetic. The different aspects of floating-point programs, the impact of this representation in computer programs and the representations and rounding errors is discussed in [13]. By showing various examples, Goldberg concluded that how the computer builders can support floating-point in an efficient way. Patrick Cousot and Radhia Cousot [5, 6] have concluded that the abstract interpretation provides the theory of approximation related to the automate formal methods. Titolo et al. [19] have presented a framework of abstract interpretation for analysis of round-off errors in floating-point programs. A sound approximation is done for the error accumulated over different floating-point computation path.

3. Related works

Floating-point programs can use abstract domains to set bounds on the ranges of variables. One of the main work in this area is the tool Fluctuat [10] which quantifies the round-off errors. For range computations, Gappa tool [11] generates a proof checkable by interactive theorem prover.

Precimonious [18], a program analysis tool developed by Rubio-Gonzales et al for tuning the precision of floating-point programs. It recommends a type configuration

for program variables to improve the performance in floating-point programs. It also evaluated various numerical programs to show the effectiveness.

FPTuner [3] uses the formal analysis via Symbolic Taylor Expansions, and error analysis based on interval functions to approach precision. It generates and solves a quadratically constrained quadratic program to obtain a precision-annotated version of a given expression.

In the paper [9], Darulova et al presents a programming model where a compilation algorithm that generates a finite-precision implementation that is guaranteed to meet the desired precision with respect to real numbers.

In our approach, we determine the minimal accuracy required for the inputs to achieve a certain level of accuracy in precisions in the results and the intermediary operations.

4. Abstract semantics

4.1. Abstract domain

Let β_p be the set of all binary representations of floating-point numbers with mantissa of length p. Basically an element $x \in \beta_p$ is defined by the following representation:

$$x = \pm b_0.b_1b_2 \dots b_p \times 2^e$$

where $b_0, b_1, ...b_p$ is the mantissa, $p \in \mathbb{N}$ and exponent range of floating-point $e \in [e_{min}, e_{max}]$ [15].

Following the IEEE754 Standard [15], in the binary 64 format, we have $52 + 1$ bits of mantissa, $e_{min} = -1022$ and $e_{max} = +1023$. Also we define:

$$\beta = \bigcup_{p \in \mathbb{N}} \beta_p$$

where β_p is a set of all floating-point numbers with different length.

The IEEE754 Standard also defines some rounding modes [15], towards $+\infty$ *(rounding up or ceiling)*, $-\infty$ *(rounding down or floor)*, 0 *(directed rounding towards zero)* and **to the nearest** *(if the number falls midway, it is rounded to the nearest value with an even least significant digit)*. Consider a normal float value 12.5, the different rounding modes results as 13.0, 12.0, 12.0, 12.0/13.0(ties to even/ties away from zero) respectively. Let us write $\circ_{p,+\infty}$, $\circ_{p,-\infty}$, $\circ_{p,0}$ and $\circ_{p,\sim}$ the rounding functions which round arbitrary numbers to numbers in precision p following the desired mode. The IEEE754 Standard defines the semantics of the elementary operations by:

$$x \circledast_{p,r} y = \circ_{p,r}(x * y)$$

where $\circledast_{p,r}$ denotes floating-point operations (such as $+$, $-$, $\times$ or $\div$) computed using the rounding mode r and precision p and where $*$ denotes an exact operation.

Let I_p be the set of all intervals for floating-point numbers with a specific precision p. An element $i^\sharp \in I_p$, denoted $i^\sharp = [f_1, f_2]_{[p]}$, is defined by two floating-point numbers and a mantissa length p. We have:

$$I_p \ni [f_1, f_2]_p = \{f \in \beta_p : f_1 \leq f \leq f_2\} \text{ and } I = \bigcup_{p \in \mathbb{N}} I_p$$

Our abstract domain is $\langle I, \sqsubseteq, \sqcup, \sqcap, \bot_I, \top_I \rangle$. The elements are ordered by:

$$[a, b]_p \sqsubseteq [c, d]_q \Longleftrightarrow [a, b] \subseteq [c, d] \text{ and } p \leq q$$

where p and q and precisions of the intervals $[a, b]$ and $[c, d]$ respectively. In other words, $[a, b]_p$ is more precise than $[c, d]_q$ if it is a smaller interval with a greater accuracy.

The least upper bound and greatest lower bound [12] is referred as join ($\sqcup$) and meet ($\sqcap$) operations and are defined by:

$$\begin{gathered}[a, b]_p \sqcup [c, d]_q = [\circ_{r,-\infty}(u), \circ_{r,+\infty}(v)]_r \\ \text{with } r = \min\{p, q\},\ [u, v] = [a, b] \cup [c, d]\end{gathered}$$

and

$$\begin{gathered}[a, b]_p \sqcap [c, d]_q = [u, v]_r \\ \text{with } r = \max\{p, q\},\ [u, v] = [a, b] \cap [c, d]\end{gathered}$$

In addition, we have:

$$\bot_I = \emptyset_{+\infty} \text{ and } \top_I = [-\infty, +\infty]_0$$

where $\bot$ is the least element referred as bottom and $\top$ is the greatest element referred as top.

We define $\alpha : \wp(\beta) \to I$, the abstraction function such as for a set of floating--point numbers B with different precisions p_i, $1 \leq i \leq n$, we associate a value of I. Let $x_{min} = \min(B)$, $x_{max} = \max(B)$ and $p = \min\{q \ :\ x \in B \text{ and } x \in \beta_q\}$ the minimal precision in B. In other words, we have:

$$\begin{gathered}\alpha(B) = [\circ_{p,-\infty}\big(\min(B)\big), \circ_{p,+\infty}\big(\max(B)\big)]_p \\ \text{where } p = \min\{q \ :\ B \cap \beta_q \neq \emptyset\}\end{gathered} \tag{1}$$

Consider the example, where we take four floating-point numbers 2.25, 2.5, 2.875, 2.890625 with 4, 5, 6 and 7 bits of accuracy, respectively. We have:

$$\begin{gathered}B = \{2.25_4, 2.5_5, 2.875_6, 2.890625_7\} \\ \min(B) = 2.25, \quad \max(B) = 2.890625\end{gathered} \tag{2}$$

and

$$\alpha(B) = [2.25, 3.0]_4$$

The minimal accuracy is 4, $\circ_{4,-\infty}(2.25) = 2.25$ and $\circ_{4,+\infty}(2.890625) = 3.0$.

Let $\gamma : I \rightarrow \wp(\beta)$, is the inverse of α and $i^\sharp = [a, b]_p$. The concretization function $\gamma(i^\sharp)$ is defined as:

$$\gamma(i^\sharp) = \bigcup_{q \geq p} \{x \in \beta_q \ : a \leq x \leq q\} \tag{3}$$

Let us consider the following example of equation (2) with floating-point interval with their binary representation: $[2.25, 2.890625]_6$. Here:

$$i^\sharp = [2.25, 2.891]_6$$

$\gamma(i^\sharp)$ can be represented as:

$$\begin{array}{lcl} 2.25_6 & \rightarrow & 0\ 10000000\ 001000 \\ 2.5_6 & \rightarrow & 0\ 10000000\ 100000 \\ 2.75_6 & \rightarrow & 0\ 10000000\ 001100 \\ \vdots & & \end{array}$$

Finally, using the functions of equations (1) and (3), we define the Galois connexion [8] as:

$$\langle B, \subseteq, \cup, \cap, \bot_B, \top_B \rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle I, \sqsubseteq, \sqcup, \sqcap, \bot_I, \top_I \rangle \tag{4}$$

4.2. Transfer functions

In this section, we introduce the rules that compute the precision of the nodes in a computation tree. Every program we write can be represented as control flow diagram (CFG) which depicts the program point and the state of the variables in the program at each control point. Using the CFG we can analyze the static behaviour of the program. Each control point can be considered as a node. There can be forward analysis or backward analysis or both can be done on the CFG. If we take a program point p, in a forward analysis taking into account the nodes which were preceding we can conclude the behaviour of the node at program point p. And in backward analysis we try to reason the facts from p to reach successors. There are two classes of transfer functions. We use the first for forward analysis and the second for backward analysis. For each class, we give the transfer function for the two basic operations: addition and multiplication.

4.2.1. Transfer functions for forward analysis

A forward analysis is one that for each program point computes information about the past behavior.

Transfer function for addition in forward analysis. Consider the addition of two intervals $x = [a, b]_{p1}$ and $y = [c, d]_{p2}$ with $a = s_1 \cdot m_1 \cdot 2^{e_1}$, $b = s'_1 \cdot m'_1 \cdot 2^{e'_1}$, $c = s_2 \cdot m_2 \cdot 2^{e_2}$ and $d = s'_2 \cdot m'_2 \cdot 2^{e'_2}$.

In the forward analysis, in order to estimate the precision for the result $z = x + y$ we take different cases. Let $d = e_2 - e_1$.

$$p = \begin{cases} min(p_1, p_2), \text{if } e_1 = e_2 \\ min(p_1 - 1, p_2 + d - 1), \text{if } e_1 > e_2 \\ min(p_1 + d - 1, p_2), \text{if } e_1 < e_2 \end{cases}$$

Transfer function for multiplication in forward analysis. Consider the multiplication of two intervals $x = [a, b]_{p1}$ and $y = [c, d]_{p2}$ with $a = s_1 \cdot m_1 \cdot 2^{e_1}$, $b = s'_1 \cdot m'_1 \cdot 2^{e'_1}$, $c = s_2 \cdot m_2 \cdot 2^{e_2}$ and $d = s'_2 \cdot m'_2 \cdot 2^{e'_2}$. The resultant $z_{p'}$ whose precision p' is estimated as:

$$p' = \begin{cases} min(p_1, p_2) - 1, \text{for all } e_1 e_2 \text{ cases} \end{cases}$$

4.2.2. Transfer functions for backward analysis

A backward analysis is one that for each program point computes information about the future behavior. Here, the addition and multiplication transfer functions will be the same as that of the forward analysis. Only the computations are done in the reverse order and we come to the minimal accuracy that is required for the inputs to achieve an asserted precision for the final result.

Units in first place. In the numerical analysis, the accuracy of a result is sometimes measured by the "unit in the last place (*ulp*)". Sometimes delicate error estimations in the ulp-concept have the drawback that it depends on the floating-point format and needs extra care in the underflow range [16]. So we use "unit in the first place" (*ufp*) or leading bit of a real number by:

$$0 \neq r \in \mathbb{R} \Rightarrow ufp(r) = 2^{log_2 \lfloor r \rfloor}$$

Consider x_p and y_q, where x and y are two floating-point numbers with p and q precision accuracy respectively. While adding we get result r_{pr} as follows:

$$r_{pr} = x_p + y_q$$

Let ϵ_x and ϵ_y be the error in the float-number x and y respectively. That is:

$$\epsilon_x \leq 2^{i-p} \quad \text{and} \quad \epsilon_y \leq 2^{j-q}$$

where $i = ufp(x)$ and $j = ufp(y)$. We can find the error in result as:

$$\epsilon_r = \epsilon_x + \epsilon_y$$

Precision of the the result can be obtained as (see Fig. 2):

$$pr = ufp(r) - ufp(\epsilon_r)$$

Figure 2. Precision of the result mapped using ufp

We take the same operands x_p and y_q to get the product r_{pr}. That is:

$$r_{pr} = x_p \times y_q \tag{5}$$

The error of x and y is represented as ϵ_x and ϵ_y and the resultant error is calculated as ϵ_r. Here we calculate the error in the result as:

$$\epsilon_r = (\epsilon_x \times y) + (\epsilon_y \times x) + (\epsilon_x \, \epsilon_y)$$

The precision in the result can be calculated as (see Fig. 2):

$$pr = ufp(r) - ufp(\epsilon_r)$$

Using the result and one of the operands of the addition in forward analysis, we do backward analysis and get the other operand. So we will take the operand(y_q) and result(r_{pr}) from equation (5) to reach x_p as follows:

$$x_p = r_{pr} - y_q$$

The error and *ufp* can be calculated as follows:

$$\tfrac{1}{2}\, ufp(x) \leq \epsilon_x \leq ufp(x) \tag{6}$$

$$\tfrac{1}{2}\, ufp(y) \leq \epsilon_y \leq ufp(y) \tag{7}$$

Combining the equations (6) and (7) and also referring to equation (5), we get:

$$\tfrac{1}{2}\, ufp(x) + \tfrac{1}{2}\, ufp(y) \leq \epsilon_r \leq ufp(x) + ufp(y) \tag{8}$$

We consider the left side of the equation (8):

$$\tfrac{1}{2}\, ufp(x) \leq \epsilon_r - \tfrac{1}{2}\, ufp(y)$$

$$ufp(x) \leq 2\,\epsilon_r - ufp(y)$$

As we did backward analysis with addition using the result and one of the operands of the addition in forward analysis, we are going to do backward analysis on multiplication. We will use the result (r_{pr}) and operand (y_q) from equation (2) to reach x_p. We consider $y \geq 0$.

$$x_p = r_{pr} \div y_q \tag{9}$$

Using the ufp error in equation (9), we get:

$$\tfrac{1}{2}\, ufp(x)\, y \;\leq\; \epsilon_x\, y \;\leq\; ufp(x)\, y \tag{10}$$

$$\tfrac{1}{2}\, ufp(y)\, x \;\leq\; \epsilon_y\, x \;\leq\; ufp(y)\, x \tag{11}$$

Combining equations (10) and (11), we get:

$$\tfrac{1}{4}\, ufp(x)\, ufp(y) \;\leq\; \epsilon_x\, \epsilon_y \;\leq\; ufp(x)\, ufp(y)$$

$$\tfrac{1}{2}\, ufp(x)\, y \;+\; \tfrac{1}{2}\, ufp(y)\, x \;\leq\; \tfrac{1}{4}\, ufp(x)\, ufp(y) \;\leq\; \epsilon_r$$

$$2\, ufp(x)\, y \;+\; ufp(x)\, ufp(y) \;\leq\; 4\, \epsilon_r \;-\; ufp(y)\, x$$

$$ufp(x) \;\leq \frac{4\, \epsilon_r \;-\; ufp(y)\, x}{2\, y\, ufp(y)}$$

5. A running example

In this section, we are going to show how our forward and backward static analyses work on a sample code snippet, given in Figure 3.

```
(0) a = 0.0; c = 0.25; t = 0;
(1) Assert x[t] = [0,1]_32 for all t;
while (t < t_max)
{
(2)  a = a * a;
(3)  a = a * c;
(4)  a = a + x[t];
t++;  (5)
}
(6) y = 1.5 * a + 1.0;
(7) Assert y = ⊥_16;
(8) End
```

Figure 3. Code snippet

In general, a forward analysis computes information from the beginning to the end of the program and a backward analysis computes information from the end of the program to the beginning. In order to take advantage of the backward analysis, we consider that the desired accuracy (the number of correct bits) is specified by the

user at the end of the program. So, we introduce an assertion at control-point (7), stating that, at this point, y belongs to an unknown interval with 16 bits of accuracy.

We consider abstract values $[a, b]_p$ for the static code analysis. Let $\mathbb{F}_p$ be the set of all floating-point numbers with accuracy p. Intuitively

$$a \in [a, b]_p \Leftrightarrow a \in \mathbb{F}_p \ : \ a \leq x \leq b$$

For example, in the code snippet of Figure 3, the assertion at control point (7) is written $y = [-\infty, +\infty]_{16}$. By extension, $\mathbb{F}_\infty$ denotes the set of exact numbers, i.e. the numbers with an infinite number of correct digits. Figures 4 and 5 show the detail of forward and backward analysis respectively.

(2) $a = [0.0, 0.0]_{53} \times [0.0, 0.0]_{53} = [0.0, 0.0]_{53}$
(3) $a = [0.0, 0.0]_{53} \times [0.25, 0.25]_{53} = [0.0, 0.0]_{53}$
(4) $a = [0.0, 0.0]_{53} + [0.0, 1.0]_{32} = [0.0, 1.0]_{32}$

(5) $a = [0.0, 0.0]_{53} \cup [0.0, 1.0]_{32} = [0.0, 1.0]_{32}$

(2)' $a = [0.0, 1.0]_{32} \times [0.0, 1.0]_{32} = [0.0, 1.0]_{31}$
(3)' $a = [0.0, 1.0]_{31} \times [0.25, 0.25]_{53} = [0.0, 0.25]_{31}$
(4)' $a = [0.0, 0.25]_{31} + [0.0, 1.0]_{32} = [0.0, 1.25]_{32}$

(5)' $a = [0.0, 1.0]_{32} \cup [0.0, 1.25]_{32} = [0.0, 1.25]_{32}$

(2)'' $a = [0.0, 1.25]_{32} \times [0.0, 1.25]_{32} = [0.0, 1.5625]_{31}$
(3)'' $a = [0.0, 1.5625]_{31} \times [0.25, 0.25]_{53} = [0.0, 0.390625]_{31}$
(4)'' $a = [0.0, 0.390625]_{31} + [0.0, 1.0]_{32} = [0.0, 1.390625]_{32}$

(5)'' $a = [0.0, 1.25]_{32} \bigtriangledown [0.0, 1.390625]_{32} = [0.0, 2.0]_{32}$

(2)''' $a = [0.0, 2.0]_{32} \times [0.0, 2.0]_{32} = [0.0, 4.0]_{31}$
(3)''' $a = [0.0, 4.0]_{31} \times [0.25, 0.25]_{53} = [0.0, 1.0]_{31}$
(4)''' $a = [0.0, 1.0]_{31} + [0.0, 1.0]_{32} = [0.0, 2.0]_{32}$
(5)''' $a = [0.0, 2.0]_{32}$

(6) $y = [1.5, 1.5]_{53} \times [0.0, 2.0]_{32} + [1.0, 1.0]_{53}$
$= [0.0, 3.0]_{32} + [1.0, 1.0]_{53} = [1.0, 4.0]_{33}$

Figure 4. Forward analysis using the code snippet

Forward analysis. We start with the forward analysis, given in Figure 4. First, a is the input variable and c is a floating-point constant. By default we assume that they have 53 digits of accuracy as for the Binary64 format [16]. At control point (1), an assertion states that all the elements of the array $x[t]$ belongs to the interval $[0.0, 1.0]$ and have accuracy 32. In practice, $x[t]$ could be a sensor with a limited accuracy or a sequence of values computed by another procedure with a limited accuracy. At control points (2) and (3) x is multiplied by itself and then by c. At the first iteration, this lets a unchanged, since $a = [0.0, 0.0]_{53}$. Next, at point (4), a is incremented by $x[t]$ which results in the value $[0.0, 0.0]_{53} + [0.0, 1.0]_{32} = [0.0, 1.0]_{32}$.

```
(-7)  y = [1.0,4.0]_33 ∪ ⊥_16 = [1.0,4.0]_16
(-7)  c = [0.25,0.25]_53

(-6)  a = (([1.5,1.5])^-1) × ([1.0,4.0]_16 − [1.0,1.0]_53)
= [0.0,2.0]_15

(-5)  a = [0.0,2.0]_32 ∩ [0.0,2.0]_15 = [0.0,2.0]_15
(-5)  c = [0.25,0.25]_53
(-4)  a = [0.0,2.0]_15(+ [0.0,1.0]_32))^-1 = [0.0,1.0]_14
(-4)  x[t] = [0.0,2.0]_15(+ [0.0,1.0]_14)^-1 = [0.0,1.0]_15
(-3)  a = [0.0,1.0]_14(× [0.0,0.25]_53)^-1 = [0.0,4.0]_14
(-3)  c = [0.0,1.0]_14(× [0.0,4.0]_14)^-1 = [0.0,0.25]_15
(-2)  a = [0.0,4.0]_14(× [0.0,2.0]_32)^-1 = [0.0,2.0]_14
(-2)  a = [0.0,4.0]_14(× [0.0,2.0]_14)^-1 = [0.0,2.0]_15

(-5)'  a = [0.0,2.0]_15 ∪ [0.0,2.0]_15 = [0.0,2.0]_15
(-5)'  a = [0.25,0.25]_53 ∪ [0.25,0.25]_15 = [0.25,0.25]_15
(-5)'  x[t] = [0.0,1.0]_32 ∪ [0.0,1.0]_15 = [0.0,1.0]_15
(-4)'  a = [0.0,2.0]_15(+ [0.0,1.0]_15))^-1 = [0.0,1.0]_14
(-4)'  x[t] = [0.0,2.0]_15(+ [0.0,1.0]_14)^-1 = [0.0,1.0]_15
(-3)'  a = [0.0,1.0]_14(× [0.0,0.25]_15)^-1 = [0.0,4.0]_14
(-3)'  c = [0.0,1.0]_14(× [0.0,4.0]_15)^-1 = [0.0,0.25]_15
(-2)'  a = [0.0,4.0]_14(× [0.0,2.0]_15)^-1 = [0.0,2.0]_14
(-2)'  a = [0.0,4.0]_15(× [0.0,2.0]_15)^-1 = [0.0,2.0]_15

(-1)  a = [0.0,2.0]_15
(-1)  c = [0.0,0.25]_15
(-1)  x[t] = [0.0,1.0]_15
```

Figure 5. Backward analysis using the code snippet

After execution of the first iteration, the control-flow comes back to point (2), at the second iteration of the loop, denoted point $(2)'$ in Figure 4. As usually in abstract interpretation, in order to over-approximate the whole concrete executions of the code, at point $(2)'$, the value given to a is the join of a at points (2) and (4). The second iteration is then executed, a is squared, multiplied by c and finally incremented by $x[t]$, yielding $[0.0, 1.25]_{32}$. After the addition, the new value of a has one more correct digit than the former. If $a \in [0.0, 1.0]_{32}$ then a has 32 correct digits and the error between a and the exact value $\mathbf{a}$ that it represents is bounded by:

$$|a - \mathbf{a}| < 2^{-32}$$

and the error on $a \times c$ is bounded by 2^{-33}. Figure 4 shows one more iteration, starting at point $(2)''$ and terminating at point $(4)''$.

Here, we observe that iterations may keep continuing, so at control point $(2)'''$, we do an over-approximation using a widening operation. In the theory of abstract interpretation [6], the widening operators over-approximate over the abstract iterations.

The chain of abstract iterations can go potentially forever. To enforce termination to this iteration we use a widening operator denoted $\bigtriangledown$ by [5]. Widening/Narrowing approach can improve the precision and the speed of convergence of the analysis significantly. We use a staged widening which returns the next power of 2 when the bounds of the intervals are increasing. This over-approximates a with the value $[0, 2]_{32}$ at control point $(2)'''$. Then, one more iteration is run and a fixed-point is reached. Finally, the abstract value of y is evaluated at point (6).

Backward analysis. Backward analysis we start from control point (7) in a backward flow denoted (-7) in Figure 5 in order to indicate that we are in the backward phase, we benefit from the assertion $y = \bot_{16}$ and start from the result of the forward analysis which is $y = [1.0, 4.0]_{33}$. The join of both predicates yields the other operand $y = [1.0, 4.0]_{33} \cup \bot_{16} = [1.0, 4.0]_{16}$. Next, at point (6), the statement $y = 1.5 \times a + 1$ is is evaluated backward as $a = 1.5^{-1}y - 1$, yielding $a = [0.0, 2.0]_{15}$. At control point(-4) and (-4)', the transfer function for addition will be used with the case $e1 > e2$ and the case $e1 < e2$ respectively. At control points (-3) and (-3)', the multiplication transfer function is used.

After two iterations of the backward analysis, a fixed-point is reached. As a final result, we have inferred that in order to obtain a result y with 16 correct bits, the input a must have at least 15 correct bits at the beginning of the execution. In addition, c and the elements of the array a must also have at least 15 correct bits.

This information is obtained by combining the forward and backward analyses. A simple forward analysis could only infer this property by successive tries, by assigning iteratively arbitrary accuracies to a, c and x at the beginning of the code and by observing the results until the desired accuracy on the result is observed. Thus Backward analysis helps in finding the minimal accuracy required for the inputs to get the desired accuracy for the output.

6. Conclusions

The static analysis helps to identify and taking necessary steps in avoiding different flaws that can happen in computations with floating-point values. In this paper, we are addressing one of the open areas in the field of static analysis in safety critical problems to decide on the minimal accuracy required in inputs for achieving the desired accuracy in outputs. We have presented a combined forward and backward analysis of floating- point programs. An abstract domain for intervals with floating-point values has been defined and the transfer functions for basic mathematical operations like addition and multiplication are also defined. As future work, we will be extending the analysis for other mathematical operations like division and bisection of floating-point intervals. Also a prototype for demonstrating the abstraction and concretization on the domain is being implemented.

References

[1] Ariane 501 Inquiry Board Report. Tech. rep., European Space Agency, 1996. https://www.esa.int/For_Media/Press_Releases/Ariane_501_-_Presentation_of_Inquiry_Board_report.

[2] Blanchet B., Cousot P., Cousot R., Feret J., Mauborgne L., Miné A., Monniaux D., Rival X.: A static analyzer for large safety-critical software, *SIGPLAN Notices*, vol. 38(5), pp. 196–207, 2003, https://doi.org/10.1145/780822.781153.

[3] Chiang W.F., Baranowski M., Briggs I., Solovyev A., Gopalakrishnan G., Rakamarić Z.: Rigorous floating-point mixed-precision tuning, *SIGPLAN Notices*, vol. 52(1), pp. 300–315, 2017, https://doi.org/10.1145/3093333.3009846.

[4] Cortesi A.: Widening Operators for Abstract Interpretation. In: *2008 Sixth IEEE International Conference on Software Engineering and Formal Methods*, pp. 31–40, 2008. https://doi.org/10.1109/SEFM.2008.20.

[5] Cousot P.: Abstract interpretation: Achievements and perspectives. In: *Proceedings of the SSGRR 2000 Computer and eBusiness International Conference*, Compact Disk Paper 224 and Electronic Proceedings. Scuola Superiore G. Reiss Romoli, Italy, 2000.

[6] Cousot P., Cousot R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL'77, pp. 238–252, ACM, New York, 1977. https://doi.org/10.1145/512950.512973.

[7] Cousot P., Cousot R.: Abstract interpretation and application to logic programs, *The Journal of Logic Programming*, vol. 13(2), pp. 103–179, 1992, https://doi.org/10.1016/0743-1066(92)90030-7.

[8] Cousot P., Cousot R.: A galois connection calculus for abstract interpretation, *SIGPLAN Notices*, vol. 49(1), pp. 3–4, 2014, https://doi.org/10.1145/2578855.2537850.

[9] Darulova E., Kuncak V.: Sound compilation of reals, *SIGPLAN Notices*, vol. 49(1), pp. 235–248, 2014, https://doi.org/10.1145/2578855.2535874.

[10] Delmas D., Goubault E., Putot S., Souyris J., Tekkal K., Védrine F.: Towards an Industrial Use of FLUCTUAT on Safety-Critical Avionics Software. In: M. Alpuente, B. Cook, C. Joubert (eds.), *Formal Methods for Industrial Critical Systems*, pp. 53–69, Springer, Berlin, Heidelberg, 2009.

[11] Dinechin de F., Lauter C., Melquiond G.: Certifying the Floating-Point Implementation of an Elementary Function Using Gappa, *IEEE Transactions on Computers*, vol. 60(2), pp. 242–253, 2011, https://doi.org/10.1109/TC.2010.128.

[12] Gange G., Navas J.A., Schachte P., Søndergaard H., Stuckey P.J.: Abstract Interpretation over Non-lattice Abstract Domains. In: Logozzo F., Fähndrich M. (eds.), *Static Analysis*, pp. 6–24, Springer, Berlin, Heidelberg, 2013.

[13] Goldberg D.: What every computer scientist should know about floating-point arithmetic, *ACM Computing Surveys*, vol. 23(1), pp. 5–48, 1991. https://doi.org/10.1145/103162.103163.

[14] Goubault E.: Static Analyses of the Precision of Floating-Point Operations. In: *Proceedings of the 8th International Symposium on Static Analysis, SAS '01*, pp. 234–259, Springer-Verlag, London, 2001. http://dl.acm.org/citation.cfm?id=647170.718304.

[15] IEEE Standard for Floating-Point Arithmetic. In: *IEEE Std 754-2008*, pp. 1–70, 2008. https://doi.org/10.1109/IEEESTD.2008.4610935.

[16] Muller J.M.: *On the definition of ulp(x)*, Research Report, RR-5504, LIP RR-2005-09, INRIA, LIP, pp. 16, 2005. https://hal.inria.fr/inria-00070503.

[17] *PATRIOT MISSILE DEFENSE. Software Problems Led to System Failure at Dhahran, Saudi Arabia*, Report to the Chairman, Subcommittee on Investigations and Oversight, Committee on Science, Space, and Technology, House of Representatives, United States General Accounting Office, Information Management and Technology Division, 1992. http://www-users.math.umn.edu/~arnold//disasters/GAO-IMTEC-92-96.pdf.

[18] Rubio-González C., Nguyen C., Nguyen H.D., Demmel J., Kahan W., Sen K., Bailey D.H., Iancu C., Hough D.: Precimonious: tuning assistant for floating-point precision. In: *SC '13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pp. 1–12, 2013. https://doi.org/10.1145/2503210.2503296.

[19] Titolo L., Feliú M.A., Moscato M., Muñoz C.A.: An Abstract Interpretation Framework for the Round-Off Error Analysis of Floating-Point Programs. In: I. Dillig, J. Palsberg (eds.), *Verification, Model Checking, and Abstract Interpretation*, vol. 10747, pp. 516–537, Springer International Publishing, Cham, 2018.

Affiliations

M.G. Thushara
Amrita School of Engineering, Department of Computer Science and Applications, Amrita Vishwa Vidyapeetham, Amritapuri, India, e-mail: thusharamg@am.amrita.edu

K. Somasundaram
Amrita School of Engineering, Department of Mathematics, Coimbatore, Amrita Vishwa Vidyapeetham, India, e-mail: s_sundaram@cb.amrita.edu,
ORCID ID: https://orcid.org/0000-0003-2226-1845

Received: 28.08.2019
Revised: 25.12.2019
Accepted: 15.01.2020