Termination analysis of floating-point programs using parameterizable rational approximations

Fonenantsoa Maurica Andrianampoizinimaro, Frédéric Mesnard, Etienne Payet

Termination analysis of floating-point programs using parameterizable rational approximations

Fonenantsoa Maurica
Université de La Réunion
LIM
France
fonenantsoa.maurica@gmail.com

Frédéric Mesnard
Université de La Réunion
LIM
France
frederic.mesnard@univ-reunion.fr

Étienne Payet
Université de La Réunion
LIM
France
etienne.payet@univ-reunion.fr

ABSTRACT

Analysis of floating-point programs is a topic that received an increasing attention the past few years. However, only very few works have been done regarding their termination analysis. We address that problem in this paper. We present a technique that takes advantage of the already existing works on termination analysis of rational programs. Our approach consists in translating the floating-point programs into rational ones by means of sound approximations. We approximate the floating-point expressions using piecewise linear functions. Our approximation differs from the already existing ones in the sense that it can be as precise as needed.

CCS Concepts

•**Software and its engineering** → **Formal methods; Software verification;**

Keywords

Termination analysis, Floating-point arithmetic, Linearization

1. INTRODUCTION

Termination analysis of programs is a hot research topic that has already produced many techniques. Some of the most innovative ones are the synthesis of linear ranking functions for simple loops [15], the generalization to eventual linear functions [1], the polyranking [6], and Chen's algorithm [7].

Unfortunately, none of these techniques can be used for the termination analysis of floating-point programs since floating-point computations are highly non-linear due to the rounding errors. To our knowledge, there is only very limited work on that topic. [16] extends the adornments-based approach. [8] reduces the termination problem to a second-order satisfiability problem.

This paper develops a new technique that helps addressing that lack of work. Its main advantage is that, through the use of an innovative rational approximation, it transposes the termination analysis of floating-point loops into termination analysis of rational loops, which is already very well covered in the literature.

The rest of the paper is organized as follows. First, we quickly introduce the basics of floating-point arithmetic in Section 2. Then, we give the details of our rational approximation in Section 3. We end by presenting its application to termination analysis in Section 4.

2. FLOATING-POINT ARITHMETIC

A real number $x \in \mathbb{R}$ is approximated in machine by a floating-point number $\hat{x} \in \mathbb{F}$. The IEEE-754 standard defines the floating-point arithmetic. A floating-point number is represented by the triplet (s, m, e) such that:

$$\hat{x} = (-1)^s \cdot \beta^e \cdot m$$

where $s \in \{0, 1\}$ is the *sign*, $\beta \in \{2, 10\}$ is the *radix*, $e \in [e_{min}, e_{max}]$ is the *exponent*, the fractional number $m = m_0.m_1m_2...m_{p-1}$ is the *significand* and is written in p digits where p is called *precision*.

We call *machine epsilon* ϵ the distance between 1 and the floating-point number following 1. We call *unit in the last place*, or *ulp*, of $\hat{x}$ the value the least significant digit represents: $ulp(\hat{x}) = \epsilon \cdot \beta^e$.

If $\hat{x}$ is such that $|\hat{x}| \geq \beta^{e_{min}}$, $\hat{x}$ is called a *normal number*. Otherwise, $\hat{x}$ is called a *subnormal number*. We call s_{min} the smallest positive subnormal, n_{min} the smallest positive normal and n_{max} the biggest positive normal.

IEEE-754 requires floating-point arithmetic operations to be correctly rounded. That is to say, giving a rounding mode $\square$, a real arithmetic operation $\star$, its floating-point equivalent $\circledast$ and 2 floating-point numbers $\hat{x}_1, \hat{x}_2$, the following holds: $\hat{x}_1 \circledast \hat{x}_2 = \square(\hat{x}_1 \star \hat{x}_2)$.

Apart from the format of the representation and the rounding modes, IEEE-754 also defines special numbers and exception handling. However, for simplicity, we will only consider the *rounding to nearest ties to even*. Moreover, we will

ACM 978-1-4503-3738-0.
http://dx.doi.org/10.1145/0000000.0000000

consider that there is no special value in our representation and that no exception will occur in our computations, notably no overflow.

As illustrative example, consider the extremely simple type *myfloat* presented in Figure 1 where $\beta = 10, p = 2, e_{min} = 0, e_{max} = 1$.

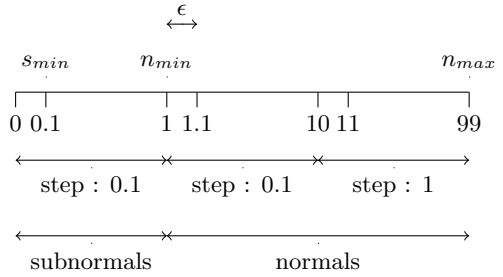


Figure 1: A personalized floating-point type, *myfloat*. Symmetry to the origin for the negatives.

A more complete introduction to floating-point computations can be found in [10].

3. RATIONAL APPROXIMATION

The common method for approximating $\hat{x}$ is to use the absolute error and the relative error. We recall the definition of the absolute error $\mathscr{A}(x) = |x - \hat{x}|$ and the relative error $\mathscr{R}(x) = \frac{|x-\hat{x}|}{|x|}$, more details can be found in [11, Section 2.1]. The linear expressions of these errors vary according to the range of the floating-point numbers that are studied, resulting in piecewise linear approximations.

To our knowledge, literature mainly approximates by distinguishing the range of the normals from the range of the subnormals. Such is the case for example of [4, 5] where the absolute error is used for the subnormals and the relative error for the normals. Sometimes a mix of the two errors is used, resulting in a single approximation for both the subnormals and the normals, like in [2, 14].

Though these approximations are correct, they may be too imprecise to be of use. Thus, more precise approximations are needed. However, the optimal bound for the absolute error has long been known and the optimal bound for the relative error has very recently been proved in [12]. Hence, the traditional approximations cited above cannot be refined anymore.

The novelty of our approximation is threefold. Firstly, it can be as precise as needed, until the equivalence between the floating-point expression and its approximation is reached. Secondly, it unifies the different types of current linear approximations (approximation by absolute error, by relative error, by combination of both) under a single framework in which they are classified hierarchically. Thirdly, it gives another proof for the optimality of the error bound of floating-point roundings.

3.1 Approximation of $\hat{x}$ with parameterizable precision

For easily understanding our idea, let us study the approximation of a real x by $\hat{x} = o(x)$ for the particular case of the type *myfloat* when the rounding to nearest ties to even is used. The rounding function o is exactly defined as follows:

$$o : \mathbb{R} \rightarrow \mathbb{F}_{myfloat}$$

$$x \mapsto o(x) = \hat{x} = \begin{cases} 99 & 98.5 < x < 99.5 \\ 98 & 97.5 \leq x \leq 98.5 \\ 97 & 96.5 < x < 97.5 \\ ... & \\ 0 & -0.05 \leq x \leq 0.05 \\ ... & \\ -99 & -99.5 < x < -98.5 \end{cases}$$

The key point is to notice that o is already a piecewise linear function, a piecewise constant function actually. As such, o is not different from the approximations by absolute/relative error which are also piecewise linear functions.

Thus, in the same way we can approximate floating-point arithmetic operations using the absolute/relative error approximations, we can also approximate using the exact definition of the rounding function, which will lead to the equivalence between the initial expression and its approximation.

However, the number of pieces used in the exact definition equals the number N of the floating-point possible values, $N = 2 \cdot \beta^{p-1} \cdot (e_{max} - e_{min} + 2) - 1$ for our case, which is a ridiculously big number. Hence, even if it may be of interest theoretically, the exact definition is hardly useful practically.

Here comes the idea of linear approximation with parameterizable precision. It can be easily understood that no piecewise linear function with a number of pieces fewer than N can exactly define $\hat{x}$. Approximations are used instead. Reducing the number of pieces will strictly reduce the precision of the approximation while increasing the number of pieces will strictly increase it.

The problem of approximating *optimally* $\hat{x}$ using linear functions defined in k pieces formally consists of finding the upper approximation function μ_k and the lower approximation function ν_k such that:

$$\begin{cases} minimize(\int_X (\mu_k(x) - \hat{x})dx) \\ \mu_k(x) = \{a_i \cdot x + b_i, x \in X_k^i\}_{1 \leq i \leq k} \\ \hat{x} \leq \mu_k(x) \end{cases} \quad (1)$$

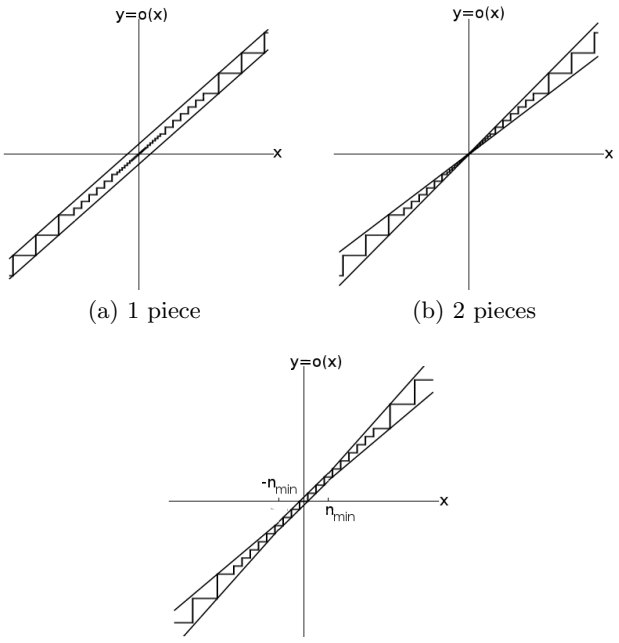
and

$$\begin{cases} minimize(\int_X (\hat{x} - \nu_k(x))dx) \\ \nu_k(x) = \{c_i \cdot x + d_i, x \in X_k^i\}_{1 \leq i \leq k} \\ \nu_k(x) \leq \hat{x} \end{cases} \quad (2)$$

where $X = [-n_{max}, n_{max}]$ and $\{X_k\}$ a partition of k elements of X that is to be found.

Optimal means that if we measure the precision of an approximation with the surface between the upper and lower approximation functions, then no approximation with greater precision can be found for the given number of pieces.

Failing at finding a general solution for (1) and (2), we consider a simplified version of the problem in which the partitioning is already known. We will now give the solutions of that simplified version. Though the optimality of our partitioning has yet to be proved, the property of increasing



(a) 1 piece (b) 2 pieces

(c) 3 pieces

Figure 2: Common piecewise linear approximations of $\hat{x}$

precision until the equivalence is preserved.

For brevity, we only present the solutions for the simplified version of (1), that is to say the value of the upper approximation function μ_k. We hope that the simplified version is clear enough so that the reader is able to solve it by himself/herself.

For $k = 1, 2, 3$ we respectively use the partitioning:

$\{X_1\} = \{[-n_{max}, n_{max}]\}$
$\{X_2\} = \{[-n_{max}, 0[, [0, n_{max}]\}$
$\{X_3\} = \{[-n_{max}, -n_{min}[, [-n_{min}, n_{min}],]n_{min}, n_{max}]\}$

We respectively find the upper linear approximation function:

$$\mu_1(x) = x + \mathscr{A}^o, x \in [-n_{max}, n_{max}]$$

$$\mu_2(x) = \begin{cases} x \cdot (1 + \mathscr{R}_n^o) + \mathscr{A}_s^o & x \in [0, n_{max}] \\ x \cdot (1 - \mathscr{R}_n^o) + \mathscr{A}_s^o & x \in [-n_{max}, 0[\end{cases}$$

and

$$\mu_3(x) = \begin{cases} x \cdot (1 + \mathscr{R}_n^o) & x \in]n_{min}, n_{max}] \\ x + \mathscr{A}_s^o & x \in [-n_{min}, n_{min}] \\ x \cdot (1 - \mathscr{R}_n^o) & x \in [-n_{max}, -n_{min}[\end{cases}$$

where $\mathscr{A}^o = \frac{\epsilon \cdot \beta^{e_{max}}}{2}$, $\mathscr{A}_s^o = \frac{\epsilon \cdot \beta^{e_{min}}}{2}$, and $\mathscr{R}_n^o = \frac{\epsilon}{2+\epsilon}$.

These are the common approximations, as used in [2, 4, 5, 14] and as illustrated in Figure 2(a), 2(b) and 2(c). $k = 1$ is the common approximation by absolute error. $k = 2$ is the common mixed use of the absolute error and the relative error. $k = 3$ is the common use of the relative error for the normals and the absolute error for the subnormals.

It is interesting to notice that the proof for the optimal bound for the relative error recently given in [12] can be retrieved by solving the simplified version of (1):

$$\begin{cases} minimize(\int_{n_{min}}^{n_{max}} (\mu(x) - \hat{x}) dx) \\ \mu(x) = a \cdot x + b \\ \mu(x) \geq \hat{x} \\ x \geq n_{min} \end{cases}$$

which will give $\mu(x) = x \cdot (1 + \delta), \delta = \frac{\epsilon}{2+\epsilon}$, after which the identification of δ as $\mathscr{R}_n^o$ easily follows. This result shows how far reaching our approach would be.

Using our formalization, we could prove the optimality of the common linear approximations and classified them in order of increasing precision. We can as well call them the 1-piece, 2-pieces and 3-pieces linear approximations. Now we will present approximations with higher precision.

When $k \geq 4$, we start from the approximation for $k = 3$ and, at each increase of k, we consecutively approximate the floating-point numbers with same *ulp* in decreasing order of *ulp*. In other words, we group by *ulp* from the edge to the center. Floating-point numbers $\hat{x} \in I = [\hat{x}_m, \hat{x}_M]$ with same *ulp* have a unique optimal linear upper approximation which is $x + \mathscr{A}_I^o$ where $\mathscr{A}_I^o = \frac{ulp(\hat{x})}{2}$.

For example, for the approximation with 4 pieces, we start from the approximation with 3 pieces. Then for the new set, we have the choice between a group of positive floating-point numbers and a group of negative floating-point numbers. We decide to take the positive one. This is illustrated in Figure 3(a).

For the approximation with 5 pieces, we start from the approximation with 4 pieces. Then for the new set, we have only a single choice, which is the previous group of negative floating-point numbers. This is illustrated in Figure 3(b).

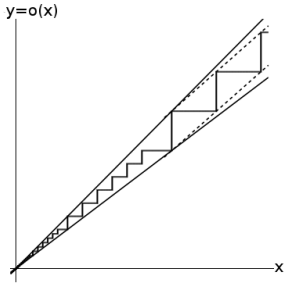
We repeat the same process for $k \geq 6$ until k is such that all floating-point numbers with same *ulp* are optimally approximated, as illustrated in Figure 3(c).

Starting from there, for the approximations with higher number of pieces, at each increase of k, we consecutively define each floating-point number exactly in decreasing order of *ulp*, in other words from the edge to the center again. The equivalence between $\hat{x}$ and its approximation is reached when $k = N$.

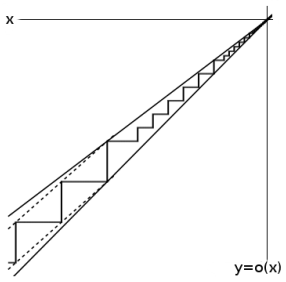
It is worth mentioning that approximating by grouping the the floating-point numbers with same *ulp* in decreasing order of *ulp* is just one possible strategy. Depending on the application and on the concrete floating-point constants used in the program, it could be beneficial for example to tighten the approximation with increasing order of *ulp* or in particular around the occurring constants.

We just presented piecewise linear approximations which are more precise than the common piecewise linear approximations by increasing the number of pieces. We will now see how to use these approximations of $\hat{x}$ to linearize arithmetic expressions.

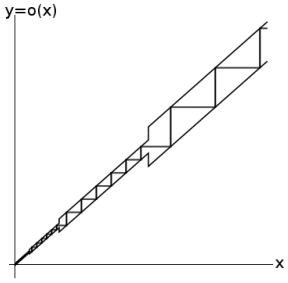
3.2 Linearization of arithmetic floating-point expressions



(a) From 3 to 4 pieces. The new set is in dashed line. Same as for 3 pieces for the negatives.



(b) From 4 to 5 pieces. The new set is in dashed line. Same as for 4 pieces for the positives.



(c) All grouped by *ulp*. Symmetry to the origin for the negatives.

Figure 3: Piecewise linear approximations of $\hat{x}$ with higher precision

Floating-point arithmetic operations are non-linear. Linearizing a floating-point arithmetic expression consists of translating the expression into a rational linear expression by means of sound approximations.

Knowing an approximation of $\hat{x}$ such that $\nu_k(x) \leq \hat{x} \leq \mu_k(x)$ and using the property of correctly rounded operations, the approximation of a floating-point arithmetic operation is straightforward: for a given arithmetic operation $\star$, we have $\nu_k(y \star z) \leq y \circledast z \leq \mu_k(y \star z)$.

When linearizing an arithmetic expression with multiple operations, we use intermediate variables to decompose the expression into elementary arithmetic operations with respect to their precedence. For example, the following expression $r = x \oplus y \odot z \ominus x$ is decomposed into:

$$\begin{cases} r = t_2 \ominus x \\ t_2 = x \oplus t_1 \\ t_1 = y \odot z \end{cases}$$

Then, it only remains to approximate each elementary operation using the previously presented piecewise linear approximations, with the desired precision. The above example will give us:

$$\begin{cases} \nu_k(t_2 - x) \leq r \leq \mu_k(t_2 - x) \\ \nu_k(x + t_1) \leq t_2 \leq \mu_k(x + t_1) \\ \nu_k(y \cdot z) \leq t_1 \leq \mu_k(y \cdot z) \end{cases}$$

Generally, once such approximation by rational expressions is obtained, the linearization is considered done even if, strictly speaking, it is not since non-linear terms, like product of two variables, may still appear in the rational expression. In those cases, the non-linear terms have to be linearized by techniques like [13]. For example, giving two rational variables $x \in [x_m, x_M]$ and $y \in [y_m, y_M]$, their product $z = x \cdot y$ will be approximated by the linear constraints:

$$\begin{cases} z - x_m y - y_m x + x_m y_m \geq 0 \\ -z + x_m y + y_M x - x_m y_M \geq 0 \\ -z + x_M y + y_m x - x_M y_m \geq 0 \\ z - x_M y - y_M x + x_m y_M \geq 0 \end{cases}$$

We will now present a concrete application of our technique through the termination analysis of loops using floating-point numbers as variables.

4. APPLICATION TO TERMINATION ANALYSIS OF FLOATING-POINT LOOPS

Our approach for analyzing termination of loops involving floating-point arithmetic consists in translating them into rational loops, and then applying the already existing termination analysis techniques to the obtained translation.

As we suppose that no overflow occurs, the result of the analysis is only valid for the initial ranges of the variables that do not lead to overflow. Abstract interpretation-based tools, like Fluctuat [9], or formal method-based tools, like Gappa [3], already exist for the determination of those ranges. The non-occurrence of overflow is an information that we can use to enrich the constraints of our approximations since for every arithmetic floating-point operation $\circledast$: $-n_{max} \leq x \circledast y \leq n_{max}$. However, for the sake of readability, we will omit these additional constraints.

Each floating-point expression is then approximated by rational ones using the technique presented in the previous section. Expressions in guard conditions are handled using intermediate variables.

We will now concretely see how to proceed through the analysis of the loop *aloop* presented in Figure 4, in which the variables x and y are of type *myfloat*. In our notation, the primed variables refer to the value of their unprimed equivalent at the end of one iteration.

```
while(x * y >= 0 & y >= -6) {
  x' := x - 0.6;
  y' := y + 0.3 + 0.3;
}
```

Figure 4: A floating-point loop, *aloop*

aloop terminates for every initial value of x and y. Indeed, for any initial value x_0, y_0 of x, y:

(i) x is always strictly decreasing,

(ii) y is strictly increasing at each iteration until reaching 10 if $-6 \leq y_0 < 10$ or y stays unchanged if $y_0 \geq 10$.

(i) and (ii) imply that if the loop runs long enough, x will become negative while y will become positive in which case the condition $x \odot y \geq 0$ will be unsatisfied.

Notice that $y'_1 = y \oplus 0.3 \oplus 0.3$ is different from $y'_2 = y \oplus 0.6$ due to the rounding errors. For example, for $y = 20$, $y'_1 = 20$ while $y'_2 = 21$.

We will now show how to analyze *aloop* using our approach. We start by decomposing all the arithmetic expressions into a succession of elementary operations. If complex expressions are encountered in the guard condition, they are put inside intermediate variables so that they can be decomposed normally. This is illustrated in Figure 5.

```
c0 := x * y;
while(c0 >= 0
      & y >= -6) {
  x' := x - 0.6;
  y' := y + 0.3 + 0.3;

  c0' := x' * y';
}
```

(a) Treatment of the guard condition

```
c0 := x * y;
while(c0 >= 0
      & y >= -6) {
  x' := x - 0.6;

  t0 := y + 0.3;
  y' := t0 + 0.3;

  c0' := x' * y';
}
```

(b) Decomposition into a succession of elementary operations

Figure 5: Preliminary operations before translation

Then, each operation is approximated by piecewise linear functions with the desired precision as presented in the previous section.

We will now show that the common approximations – the 1-piece, the 2-pieces and the 3-pieces linear approximations in our terminology – are not precise enough to prove the termination of *aloop* while the 4-pieces one is.

Using the 1-piece linear approximation, $r = y \circledast z$ is approximated by $\nu_1(y \star z) \leq r \leq \mu_1(y \star z)$ such that $\mu_1(x) = x + 0.5, -99 \leq x \leq 99$ and $\nu_1(x) = x - 0.5, -99 \leq x \leq 99$.

Applying that approximation on the transformed program in Figure 5(b), we will get the rational approximation of *aloop* in Figure 6 which uses the non-deterministic assignments `<=:` and `:<=` .

```
x*y - 0.5 <=: c0 :<= x*y + 0.5;
while(c0 >= 0
      & y >= -6) {
  x - 1.1 <=: x' :=< x - 0.1;

  y - 0.2 <=: t0 :<= y + 0.8;
  t0 - 0.2 <=: y' :<= t0 + 0.8;

  x'*y' - 0.5 <=: c0' :<= x'*y' + 0.5;
}
```

Figure 6: Rational approximation of *aloop* using the 1-piece linear approximation

The 1-piece linear approximation is precise enough to capture the decreasing of x as the obtained upper bound of x' is strictly less than x. However, it is too loose to capture the increasing to a positive number of y. Indeed, the obtained lower bound of $t0$ is less than or equal to y and that of y' is less than or equal to $t0$. Thus, with this approximation, y may decrease or may stay constant during all executions of the loop. If the initial values of x and y are negative numbers and y stays constant, the loop will run infinitely. As there are cases where the translated loop may terminate or may not, we cannot decide the termination of *aloop*.

Using the 2-pieces linear approximation, $r = y \circledast z$ is approximated by $\nu_2(y \star z) \leq r \leq \mu_2(y \star z)$ such that:

$$\mu_2(x) = \begin{cases} \frac{22}{21}x + 0.05 & x \in [0, 99] \\ \frac{20}{21}x + 0.05 & x \in [-99, 0[\end{cases}$$

and

$$\nu_2(x) = \begin{cases} \frac{20}{21}x - 0.05 & x \in [0, 99] \\ \frac{22}{21}x - 0.05 & x \in [-99, 0[\end{cases}$$

Thus, the instruction `x' := x - 0.6;` is translated as in Figure 7.

```
if (x - 0.6 >=0) {
  20*(x - 0.6)/21 - 0.05 <=: x'
  <=: 22*(x - 0.6)/21 + 0.05;
} else {
  22*(x - 0.6)/21 - 0.05 <=: x'
  <=: 20*(x - 0.6)/21 + 0.05;
}
```

Figure 7: Rational approximation of `x' := x - 0.6;` using the 2-pieces linear approximation

The other instructions are translated in a similar way.

We easily verify that the 2-pieces linear approximation is precise enough to capture the increasing to a positive number of y as the obtained lower bound of y' is strictly greater than y for $-6 \leq y \leq 0$. However, it is too loose to capture the decreasing of x for any initial value x_0 of x. Indeed, for $x_0 \geq 14$, the obtained upper bound of x' is greater than or equal to x. Thus, with this approximation, x may increase or may stay constant during all executions of the loop. If $x_0 \geq 14$ and $y_0 \geq 0$, the loop will run infinitely. As there are cases where the translated loop may terminate or may not, we cannot decide the termination of *aloop*.

The same reasoning applies for the 3-pieces linear approximation which is also precise enough to capture the increasing to a positive number of y but not the decreasing of x.

Using the 4-pieces linear approximation, $r = y \circledast z$ is approximated by $\nu_4(y \star z) \leq r \leq \mu_4(y \star z)$ such that:

$$\mu_4(x) = \begin{cases} x + 0.5 & x \in [10, 99] \\ \frac{22}{21}x & x \in]1, 10[\\ x + 0.05 & x \in [-1, 1] \\ \frac{20}{21}x & x \in [-99, -1[\end{cases}$$

and

$$\nu_4(x) = \begin{cases} x - 0.5 & x \in [10, 99] \\ \frac{20}{21}x & x \in]1, 10[\\ x - 0.05 & x \in [-1, 1] \\ \frac{22}{21}x & x \in [-99, -1[\end{cases}$$

Thus, the instruction `x' := x - 0.6;` is translated as in Figure 8.

```
if (x - 0.6 >= 10) {
 x - 1.1 <=: x' :=< x - 0.1;
} else if (1 <= x - 0.6 & x - 0.6 <= 10) {
  20*(x - 0.6)/21 <=: x' <=: 22*(x - 0.6)/21;
} else if (-1 <= x - 0.6 & x - 0.6 <= 1) {
  x - 0.65 <=: x' :<= x + 0.55;
}else {
  22*(x - 0.6)/21 <=: x' <=: 20*(x - 0.6)/21;
}
```

Figure 8: Rational approximation of `x' := x - 0.6;` using the 4-pieces linear approximation

The other instructions are translated in a similar way.

Now, the approximation is precise enough to capture both the decreasing of x and the increasing to a positive number of y. Indeed, the obtained upper bound of x' is strictly less than x and the obtained lower bound of y' is strictly greater than y for any $-6 \leq y \leq 0$. As the translated loop terminates for any initial value of x and y, so does *aloop*.

5. CONCLUSION

We have presented a new technique for analyzing termination of floating-point loops by translating them into rational ones using sound approximations. To achieve that, we use piecewise linear functions. The novelty lies in the quality of the approximations which are optimal for the chosen partitioning and can be as precise as needed by increasing or decreasing the number of pieces. That is the notion of what we call piecewise linear approximation with parameterizable precision. Unlike the few other existing techniques for termination analysis of floating-point loops, ours benefits from the very well covered termination analysis of rational and linear loops.

To some extent, the presented technique can be also applied to programs that operate on both floating-point variables and on other kind of data such as integers, arrays and more generally heap-based data-structures. To our knowledge, such a combined analysis has not yet been proposed for program termination.

Our work can be extended in many interesting ways. We can guide the partitioning by the ranges of the variables in the expression to approximate. Those ranges can be retrieved by abstract interpretation-based techniques for example. We can also use approximations with different precision for each expression to approximate instead of using a unique approximation for all the expressions.

The notion of approximation with parameterizable precision can be easily used in other topics involving floating-point arithmetic. We notably think about the topic of constraint solving over floating-point numbers which also uses rational approximations. Indeed, the more precise the approximations are, the more efficient the filtering process will be.

6. REFERENCES

[1] R. Bagnara and F. Mesnard. Eventual linear ranking functions. In *Principles and Practice of Declarative Programming*, 2013.

[2] M. S. Belaid, C. Michel, and M. Rueher. Boosting local consistency algorithms over floating-point numbers. In *Principles and Practice of Constraint Programming*. 2012.

[3] S. Boldo, J.-C. Filliâtre, and G. Melquiond. Combining Coq and Gappa for certifying floating-point programs. In *Intelligent Computer Mathematics*. 2009.

[4] S. Boldo and T. M. T. Nguyen. Hardware-independent proofs of numerical programs. In *NASA Formal Methods*, 2010.

[5] S. Boldo and T. M. T. Nguyen. Proofs of numerical programs when the compiler optimizes. *Innovations in Systems and Software Engineering*, 2011.

[6] A. R. Bradley, Z. Manna, and H. B. Sipma. The polyranking principle. In *Automata, Languages and Programming*. 2005.

[7] H. Y. Chen, S. Flur, and S. Mukhopadhyay. Termination proofs for linear simple loops. In *Static Analysis*. 2012.

[8] C. David, D. Kroening, and M. Lewis. Unrestricted termination and non-termination arguments for bit-vector programs. In *European Symposium on Programming*. 2015.

[9] D. Delmas, E. Goubault, S. Putot, J. Souyris, K. Tekkal, and F. Védrine. Towards an industrial use of Fluctuat on safety-critical avionics software. In *Formal Methods for Industrial Critical Systems*. 2009.

[10] D. Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 1991.

[11] N. J. Higham. *Accuracy and Stability of Numerical Algorithms*. Society for Industrial and Applied Mathematics, 2002.

[12] C.-P. Jeannerod and S. M. Rump. On relative errors of floating-point operations: optimal bounds and applications. Preprint, 2014.

[13] G. P. McCormick. Computability of global solutions to factorable nonconvex programs: Part I – Convex underestimating problems. *Mathematical Programming*, 1976.

[14] A. Miné. Relational abstract domains for the detection of floating-point run-time errors. In *European Symposium on Programming*. 2004.

[15] A. Podelski and A. Rybalchenko. A complete method for the synthesis of linear ranking functions. In *Verification, model checking, and abstract interpretation*, 2004.

[16] A. Serebrenik and D. De Schreye. Termination of floating-point computations. *Journal of Automated Reasoning*, 2005.