

On termination of programs with real numbers computations

Alexander Serebrenik
Danny De Schreye

Report CW 331, May 2002



Katholieke Universiteit Leuven
Department of Computer Science

Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

On termination of programs with real numbers computations

Alexander Serebrenik

Danny De Schreye

Report CW 331, May 2002

Department of Computer Science, K.U.Leuven

Abstract

Numerical computations form an essential part of almost any real-world program. Traditional approaches are restricted domains isomorphic to \mathcal{N} , more recent works study termination of integer computations. Termination of computations involving real numbers is cumbersome and counter-intuitive due to rounding errors and implementation conventions. We present a novel technique that allows us to prove termination of such computations. Our approach extends the previous work on termination of integer computations.

Keywords : termination analysis, numerical computation, floating-point number.

CR Subject Classification : D.1.6, D.2.4, I.2.2

1 Introduction

Numerical computations form an essential part of almost any real-world program. Clearly, in order for a termination analyser to be of practical use it should contain a mechanism for proving termination of such computations. However, this topic attracted less attention of the research community. Most of the approaches suggested so far were restricted to programs depending on integer computations only [13, 20] or computations over domains isomorphic to \mathcal{N} [18]. The reason for doing so is that, as shown by Dershowitz *et al.* [13], termination of computations with real numbers, may be contr-intuitive, i.e., the actually observed behaviour does not necessary coincide with the theoretically expected one¹. Difficulties encountered while working with the implementations of real numbers explain why some Prolog implementations, such as Open Prolog [7], do not include real-numbers arithmetics.

The reason for these difficulties is that the real numbers represented by computer are not mathematical objects, but their floating-point approximations.

Example 1. Consider the following programs.

$$\begin{aligned}(P_1) \quad & p(X) \leftarrow X > 0, X1 \text{ is } X * 0.25, p(X1). \\(P_2) \quad & q(X) \leftarrow X > 0, X1 \text{ is } X * 0.75, q(X1). \\(P_3) \quad & r(X) \leftarrow X > 0, X1 \text{ is } X - 0.25, r(X1).\end{aligned}$$

If numbers considered were real numbers, $p(1)$ and $q(1)$ would not terminate with respect to P_1 and P_2 , respectively, while $r(1)$ would terminate with respect to P_3 . However, computers do not work with the reals, but with the floating point numbers and computations are imprecise due to the rounding. For example, SICStus [22] rounds $3 \times 10^{15} - 0.25$ to 3×10^{15} , resulting in non-termination of $r(3 \times 10^{15})$ with respect to P_3 .

Despite the similarity between P_1 and P_2 the termination behaviour of $p(1)$ and $q(1)$ with respect to the programs is completely different. That is, $p(1)$ terminates with respect P_1 and $q(1)$ may not terminate with respect to P_2 . The reason is that if rounding is done to the nearest value, then for some t , $t * 0.75$ can be rounded upwards to t . On the other hand, for all s , $s * 0.25$ can never be rounded to s , since 0 is always closer to $s * 0.25$ than s .

¹ Examples provided in [13] contain cuts; however, similar behaviour can be observed if cut is excluded from the language.

The rounding to the nearest policy is the default rounding policy. However, a user can specify the rounding to be done, for example, toward zero. In this case, both $p(1)$ terminates with respect to P_1 and $q(1)$ terminates with respect to P_2 . \square

Example 1 shows that termination depends on the domain of the computation. However, in the early days of computing the domain of the floating point numbers was completely depending on the actual implementation, making the analysis almost impossible. To solve this problem a number of standards were suggested. As the following example hints, the existing situation, discussed in Section 3, is still not free from anomalies.

Example 2. As we have seen in the previous example, if the rounding to the nearest value is assumed, $q(1)$ does not terminate with respect to P_2 . The following program P_4 is logically equivalent to P_2 .

$$(P_4) \quad q(X) \leftarrow 1 + X > 1, X1 \text{ is } X * 0.75, q(X1).$$

In general, one might expect $1 + X > 1$ to be equivalent to $X > 0$ and $q(1)$ to be non-terminating with respect to P_4 in the same way as it does not terminate with respect to P_2 . However, $q(1)$ turns out to be terminating with respect to P_4 !

The reason for this is that the precision of floating-point numbers near 0 is much higher than near 1. Thus, there exist real numbers r , such that $r > 0$ holds, but $1 + r$ is computed to be equal to 1. When such a number is computed, the test $1 + X > 1$ fails and the computation terminates. \square

Even more surprising behaviour can be observed if non-numerical values are considered. Those are required by the standards to make the arithmetic functions total. For example, $1.0/0.0$ is defined to be $+\infty$, while $0.0/0.0$ to be *nan*. Moreover, zero is required to be signed, meaning that there should be two values representing zero: 0.0 and -0.0 . The values should be numerically equal, that is, $0.0 ::= -0.0$ should hold. As the following example illustrates, these values can result in an extremely contrived behaviour.

Example 3. Consider the following program:

$$\begin{aligned} r(X) &\leftarrow Y \text{ is } 1/X, s(Y). \\ s(X) &\leftarrow X > 0, X1 \text{ is } X + 1, s(X1). \end{aligned}$$

We would like to study execution of $r(0.0)$ and of $r(-0.0)$ with respect to this program. Some Prolog implementations, such as the MasterProLog [15] and the *iso*-mode of SICStus Prolog [22] report an error upon zero division.

On the other hand, most of the Prolog implementations (CIAO Prolog [8], ECLⁱPS^e [4] and the default mode of SICStus Prolog [22]) produce $s(+\infty)$ for $r(0.0)$ and $s(-\infty)$ for $r(-0.0)$. The order relationship $>$ is defined on the infinity values to satisfy $+\infty > r > -\infty$ for all floating-point numbers r . Computing $s(+\infty)$, thus, results in non-termination, while computing $s(-\infty)$ terminates. Thus, surprisingly, there are two values t_1 and t_2 , such that $t_1 =:= t_2$, $r(t_1)$ terminates with respect to this program and $r(t_2)$ does not terminate with respect to it. \square

Despite of these anomalies we are going to base our termination analysis, presented in Sections 4 and 5, on the standards discussed.

2 Preliminaries

We follow the standard notation for terms and atoms. A *query* is a finite sequence of atoms. Given an atom A , $rel(A)$ denotes the predicate occurring in A . $Atom_P$ denotes a set of all atoms that can be constructed from the language underlying P . The extended base B_P^E is a quotient set of $Atom_P$ modulo the variant relation. An SLD-tree constructed using the left-to-right selection rule of Prolog is called an LD-tree. A goal G *LD-terminates* for a program P , if the LD-tree for (P, G) is finite.

The following definition is similar to Definition 6.30 [5].

Definition 1. *Let P be a program and p, q be predicates occurring in it. We say that*

- p refers to q in P if there is a clause in P that uses p in its head and q in its body.
- p depends on q in P and write $p \sqsupseteq q$, if (p, q) is in the transitive closure of the relation refers to.
- p and q are mutually recursive and write $p \simeq q$, if $p \sqsupseteq q$ and $q \sqsupseteq p$.

We recall some basic notions, related to termination analysis. A *level mapping* is a function $|\cdot|: B_P^E \rightarrow \mathcal{N}$, where \mathcal{N} is the set of the naturals.

We study termination of programs with respect to sets of queries. The following notion is one of the most basic notions in this framework.

Definition 2. [11] *Let P be a definite program and S be a set of atomic queries. The call set, $Call(P, S)$, is the set of all atoms A , such that a variant of A is a selected atom in some derivation for $P \cup \{\leftarrow Q\}$, for some $Q \in S$ and under the left-to-right selection rule.*

The following definition [21] generalises the notion of acceptability with respect to a set [10, 11] by extending it to mutual recursion.

Definition 3. Let S be a set of atomic queries and P a definite program. P is acceptable with respect to S if there exists a level mapping $|\cdot|$ such that

- for any $A \in \text{Call}(P, S)$
- for any clause $A' \leftarrow B_1, \dots, B_n$ in P , such that $\text{mgu}(A, A') = \theta$ exists,
- for any atom B_i , such that $\text{rel}(B_i) \simeq \text{rel}(A)$ and for any c.a.s. σ for $\leftarrow (B_1, \dots, B_{i-1})\theta$ holds that

$$|A| > |B_i\theta\sigma|. \quad (1)$$

In [10] LD-termination was characterised in terms of acceptability.

Theorem 1. (cf. [10]) Let P be a program. P is acceptable with respect to a set S if and only if P is LD-terminating for all queries in S .

While studying termination of numerical computations relying on the real numbers, finding a level mapping that satisfies the acceptability condition as it presented in Definition 3 might be difficult. Thus, a notion of the level mapping is extended to mappings to non-negative real numbers: $|\cdot|: B_P^E \rightarrow \mathcal{R}^+$. However, requiring the usual acceptability decrease is insufficient to ensure termination, since real numbers are not well-founded. In order to obtain the well-foundedness, and therefore to prove termination, Condition (1) should be replaced with the following requirement: there exists $\varepsilon > 0$ such that $|A| - |B_i\theta\sigma| \geq \varepsilon$.

Indeed, if there is a level-mapping, satisfying Definition 3, ε can be taken equal to 1. On the other hand, if there exist a mapping to non-negative real numbers $|\cdot|$ and $\varepsilon > 0$ that satisfy the condition above then $|\cdot|_2$ defined as $|t|_2 = \left\lfloor \frac{|t|}{\varepsilon} \right\rfloor$ satisfies the acceptability condition. Thus, instead of using the acceptability condition of Definition 3, we apply the equivalent condition, which we also call *acceptability*.

In [20] we extended the previous work on a constraints based approach to termination [12] to include integer computations. The main difficulty that has to be solved was the non well-foundedness of integers. We illustrate the approach proposed by means of example and refer the reader to [20] for further details.

Example 4. We are interested in proving termination of the set of queries $\{p(n) \mid n \text{ is an integer}\}$ with respect to the following program:

$$\begin{aligned} p(X) &\leftarrow X > 1, X < 100, X1 \text{ is } -X^2, p(X1). \\ p(X) &\leftarrow X < -1, X > -100, X1 \text{ is } X^2, p(X1). \end{aligned}$$

Let $\$1$ denote the first argument. The first clause is applicable if the constraint $c_1 \equiv 1 < \$1 < 100$ holds for $p(X)$, the second one, if $c_2 \equiv -100 < \$1 < -1$ holds for $p(X)$. Thus, termination of $p(X)$ for $c_3 \equiv (\$1 \leq -100 \vee -1 \leq \$1 \leq 1 \vee \$1 \geq 100)$ is trivial. Moreover, if c_1 holds and the first clause is applied, then either c_2 or c_3 hold for the recursive call. We use this observation and specialise the program with respect to c_1, c_2 and c_3 (cf. [24]), that are called *adornments* and the program transformation is called *adorning*. In general, adornments are constructed as conjunctions of (disjunctions of) comparisons. The following program is obtained:

$$\begin{aligned}
p^{c_1}(X) &\leftarrow X > 1, X < 100, X1 \text{ is } -X^2, -100 < X1, X1 < -1, p^{c_2}(X1). \\
p^{c_1}(X) &\leftarrow X > 1, X < 100, X1 \text{ is } -X^2, \\
&\quad (X1 \leq -100; (-1 \leq X1, X1 \leq 1); X1 \geq 100), p^{c_3}(X1). \\
p^{c_2}(X) &\leftarrow X < -1, X > -100, X1 \text{ is } X^2, 1 < X1, X1 < 100, p^{c_1}(X1). \\
p^{c_2}(X) &\leftarrow X < -1, X > -100, X1 \text{ is } X^2, \\
&\quad (X1 \leq -100; (-1 \leq X1, X1 \leq 1); X1 \geq 100), p^{c_3}(X1).
\end{aligned}$$

At the next step three *natural* level mappings are defined. In general, $|p^{a_1 \wedge \dots \wedge a_n}(t_1, \dots, t_n)|$ is a weighted sum of $|p^{a_i}(t_1, \dots, t_n)|$'s. If a_i is a disjunction the corresponding level mapping is zero, otherwise $a_i = E_1 \rho E_2$ for some expressions E_1, E_2 and $\rho \in \{>, \geq\}$. Then $|p^{a_i}(t_1, \dots, t_n)|$ is $(E_1 - E_2)(t_1, \dots, t_n)$ if $E_1(t_1, \dots, t_n) \rho E_2(t_1, \dots, t_n)$ and 0 otherwise. In particular, if $a_i = \$1 > 0$, $|p^{a_i}(t_1, \dots, t_n)| = t_1$.

In our case, the natural level mappings are $|p^{c_1}(n)| = 100 - n$ if $1 < n < 100$ and 0 otherwise; $|p^{c_2}(n)| = 100 + n$ if $-100 < n < -1$ and 0 otherwise, and $|p^{c_3}| = 0$. Acceptability of the transformed program with respect to $\{p^{c_1}(n) \mid 1 < n < 100\} \cup \{p^{c_2}(n) \mid -100 < n < -1\}$ via the specified level mappings can be easily verified, implying termination of the transformed program and, thus, of the original one. \square

3 Standardisation of arithmetics

In this section we discuss standardisation of arithmetics. Traditionally, computer systems represented floating-point numbers in a normalised way, that is as $S \times 2^E$, where $1 \leq S < 2$. For example, 2.5 was represented by $S = 1.25$ and $E = 1$. The smallest positive number that can be represented in this way is, thus, 1×2^{emin} , where $emin \in \mathcal{Z}$ is the smallest exponent (\mathcal{Z} stands for the set of integers). The second smallest number is then $(1 + 2^{-p}) \times 2^{emin}$, where p is a number of digits that

can be stored. Note that the gap between zero and 1×2^{emin} is much bigger than the gap between this number and $(1 + 2^{-p}) \times 2^{emin}$. This gap near 0 resulted in a number of anomalies, such as $x = y$ being not equivalent with $x - y = 0$ or $1 \times x$ being different from x . To resolve these problems the IEEE Standard 754 [14] (also known as the IEC standard 559 for Binary Floating-Point Arithmetic for Microprocessor Systems) introduced a special set of values, called *denormalised numbers*, that are represented as $S \times 2^{emin}$, where $0 < S < 1$. The use of denormalised numbers however, results in non-equivalence of $x > 0$ and $1 + x > 1$. Thus, the ISO/IEC standard for Language Independent Arithmetic (ISO/IEC 10967) [1, 2] leaves the decision whether the denormalised numbers should be implemented to the system developer.

Since the ISO Standard for Prolog [3] follows ISO/IEC 10967 we choose to base our analysis upon it. We start by presenting the formal requirements of [1, 3].

A set of floating-point values, F , is a finite subset of the set of real numbers \mathcal{R} , characterised by five parameters: the “base” of the number system, also called the *radix*, $r \in \mathcal{Z}$ (2 in the explanation above); the number of radix digits provided by F , also called the *precision*, $p \in \mathcal{N}$, where \mathcal{N} is the set of natural numbers; the smallest and the largest exponents of F , $emin \in \mathcal{Z}$ and $emax \in \mathcal{Z}$, and a boolean flag *denorm* being *true* if F contains denormalised numbers.

Example 5. Traditionally, numbers are represented in computers in the binary format, meaning that $r = 2$. The IEEE Standard 754 requires that for the *single* format $p = 24$, $emin = -126$ and $emax = 127$ should hold and for the *double* format $p = 53$, $emin = -1022$ and $emax = 1023$ should hold. It also requires the *denorm*-flag to be set to *true*. \square

The parameters above shall satisfy: $r \geq 2 \wedge p \geq 2 \wedge p - 2 \leq -emin \leq r^p - 1 \wedge p \leq emax \leq r^p - 1$. These parameters should also satisfy: r is even $\wedge r^{p-1} \geq 10^6 \wedge (emin-1) \leq -2*(p-1) \wedge emax > 2*(p-1) \wedge -2 \leq (emin-1)+emax \leq 2$. Given specific values for the parameters, F_N , the set of *normalised* floating point values, is $\{0, \pm i * r^{e-p} \mid i, e \in \mathcal{Z}, r^{p-1} \leq i \leq r^p - 1, emin \leq e \leq emax\}$ and F_D , the set of *denormalised* floating point values, is $\{\pm i * r^{emin-p} \mid i \in \mathcal{Z}, 1 \leq i \leq r^{p-1} - 1\}$. Then, the set F is defined as $F_N \cup F_D$, if *denorm* = *true*, and as F_N , otherwise.

For convenience, the following notions are defined: the maximal floating-point number *fmax*, the minimal normalised number *fmin_N*, the minimal denormalised number *fmin_D* and F^* , the set of floating-point values ex-

tended beyond $fmax$.

$$\begin{aligned}
fmax &= \max \{z \in F \mid z > 0\} = (1 - r^{-p}) * r^{emax}, \\
fmin_N &= \min \{z \in F_N \mid z > 0\} = r^{emin-1}, \\
fmin_D &= \min \{z \in F_D \mid z > 0\} = r^{emin-p}, \\
F^* &= F \cup \{\pm i * r^{e-p} \mid i, e \in \mathcal{Z}, r^{p-1} \leq i \leq r^p - 1, e > emax\}.
\end{aligned}$$

Example 6. IEEE 754 [17] requires for the single format, $fmax$ to be 3.4×10^{38} , $fmin_N$ to be 1.2×10^{-38} and $fmin_D$ to be 1.4×10^{-45} . \square

Operations on floating point numbers are defined as following:

$$\begin{aligned}
add_F(x, y) &= result_F(add_F^*(x, y), rnd_F) \\
sub_F(x, y) &= add_F(x, -y) \\
mul_F(x, y) &= result_F(x * y, rnd_F) \\
div_F(x, y) &= result_F((x/y), rnd_F) && \text{if } y \neq 0 \\
&= zero_divisor && \text{if } y = 0
\end{aligned}$$

where add_F^* is an approximate addition, rnd_F is a floating point rounding function and $result_F$ checks the boundaries of overflow and underflow. One might imagine that functions similar to add_F^* might be required for multiplication and division. This is not the case, however, the standards require no information loss prior to rounding for these operations.

The approximate addition function was introduced due to the information loss that occurs prior to rounding in some hardware implementations. This function has to be commutative, sign symmetric and monotonic. It also has to satisfy for all $u, v, x, y \in F$ and all $i \in \mathcal{Z}$:

$$\begin{aligned}
x \leq add_F^*(u, v) \leq y &&& \text{if } x \leq (u + v) \leq y \\
add_F^*(u * r^i, v * r^i) &= add_F^*(u, v) * r^i && \text{if } u, v, u * r^i \text{ and } v * r^i \text{ are all in } F_N
\end{aligned}$$

Ideally, no information should be lost before rounding, that is, $add_F^*(u, v) = u + v$. If the last condition doesn't hold, proving termination turns out to be extremely difficult, since neither [1] nor [3] specify any requirements on the precision of add_F^* .

Function rnd_F maps real numbers to F^* and satisfies the following for all $x, y \in \mathcal{R}$ and for all $i \in \mathcal{Z}$:

$$\begin{aligned}
rnd_F(x) &= x && \text{if } x \in F^* \\
rnd_F(x) &\leq rnd_F(y) && \text{if } x < y \\
rnd_F(-x) &= -rnd_F(x) \\
rnd_F(x * r^i) &= rnd_F(x) * r^i && \text{if } |x| \geq fmin_N \text{ and } |x * r^i| \geq fmin_N
\end{aligned}$$

Example 7. IEEE Standard 754 discusses four rounding modes: round to nearest, round toward 0, round toward $-\infty$ and round toward $+\infty$. The last two rounding modes do not satisfy $rnd_F(-x) = -rnd_F(x)$, that is, they violate the ISO 10967 requirements. This one of the discrepancies between the standards despite the fact that both of them were adopted by the same institution (IEC).

The ISO 10967 standard suggests three rounding modes: round to nearest, round toward 0 and round toward $-\infty$. The latter mode contradicts the conditions above, specified by the same standard!

The only standard to resolve this problem completely is [3]. It presents only two rounding functions: round to nearest and round toward 0. These are the only two rounding functions we discuss in this paper. \square

The rounding function shall satisfy for all $x \in \mathcal{R}$

$$|rnd_F(x) - x| \leq rnd_error * r^{e_F(rnd_F(x))-p}, \quad (2)$$

where $e_F(x)$ is defined as $\lfloor \log_r |x| \rfloor + 1$ if $|x| \geq fmin_N$ and as $emin$, otherwise. The requirement that rnd_F be a rounding function implies that $rnd_error \leq 1$. If $add_F^*(x, y)$ is not identically equal to $x + y$, then rnd_error shall be defined as 1.

Depending on the fact whether rounding is done toward zero or to the nearest value, the rounding function rnd_F shall satisfy

$$\begin{aligned} |rnd_F(x)| &\leq |x| && \text{rounding is done toward zero,} && (3) \\ |rnd_F(x) - x| &\leq 0.5r^{e_F(x)-p} && \text{rounding is done to the nearest value.} && (4) \end{aligned}$$

Function $result_F$ checks the boundaries of the domain. For all $x \in \mathcal{R}$ and any rounding function $round \in (\mathcal{R} \rightarrow F^*)$, the following shall apply:

$$result_F(x, round) = \begin{cases} round(x) & \text{if } x = 0 \vee fmin_N \leq |x| \leq fmax \\ round(x) & \text{if } |x| > fmax \wedge |round(x)| = fmax \\ float_overflow & \text{if } |x| > fmax \wedge |round(x)| \neq fmax \\ round(x) \text{ or } underflow & \text{if } 0 < |x| < fmin_N \wedge |round(x)| \leq fmin_N \end{cases}$$

These requirements imply that an exception has to be reported, if the result of an arithmetic operation is *float_overflow*, *underflow*, *zero_divisor* or *undefined*. Since the exception may be trapped, a corresponding continuation value ($+\infty$, $-\infty$, *nan*) should be provided as well. IEEE 754 specifies arithmetic for microprocessors, and thus, leaves the decision whether exceptions should be trapped to a higher-level software (for example, a

Prolog system). On the other hand, ISO 10967 leaves this decision to a programmer, thus, explicitly prohibiting Prolog systems from trapping exceptions as a default policy. The following example illustrates that most of the Prolog systems violate the last requirement.

Example 8. Recall Example 3 and the execution of $r(0.0)$ with respect to it. If the ISO requirements are respected, an error should be reported, causing termination. This is the case for MasterProLog [15] and the *iso*-mode of SICStus Prolog [22].

On the other hand, if the *zero_divisor* exception is trapped by Prolog implementation, $+\infty$ is produced, resulting in a non-terminating computation. This is the case for CIAO Prolog [8], ECLⁱPS^e [4] and the default mode of SICStus Prolog [22] (*sicstus*-mode). \square

4 Examples

In this section we use the formulae of Section 3 to study the termination behaviour of the following examples.

Example 9. Consider the following program, generalising P_1 and P_2 from Example 1.

$$p(X) \leftarrow X > 0, X1 \text{ is } X * \alpha, p(X1).$$

We would like to find the values of $0 < \alpha < 1$ such that $p(t)$ terminates for all floating point numbers t . To do so we try to prove termination of $p(t)$ and collect the constraints on α we need to assume. Obviously, $p(t)$ terminates for all $t \leq 0$. Thus, in the remainder we assume $t > 0$.

As a first approximation to the level-mapping we choose, similarly to [20], $|p(t)| = t$. The level-mapping is well-defined, since $t > 0$ is assumed. If $p(t)$ is unified with the head of the clause, the value to be stored in $X1$ is $result(t\alpha, rnd_F)$, that is either $rnd_F(t\alpha)$, *floating_overflow* or *underflow*. If one of the later cases occurs execution terminates, as discussed above. Thus, we restrict our attention only to the case $result(t\alpha, rnd_F) = rnd_F(t\alpha)$. If we can find $\varepsilon > 0$, such that for any t , $t - rnd_F(t\alpha) \geq \varepsilon$, we will define a new level mapping $|\cdot|_2$, such that $|t|_2 = \left\lfloor \frac{|t|}{\varepsilon} \right\rfloor$. Then, the program will be acceptable with respect to $\{p(1)\}$ via $|\cdot|_2$.

Assume that rnd_F has the *round to nearest* property. Then $|rnd_F(t\alpha) - t\alpha| \leq 0.5r^{e_F(t\alpha)-p}$. We distinguish between the following cases:

- $|t\alpha| < fmin_N$. Then, $e_F(t\alpha) = emin$ and $r^{emin-p} = fmin_D$. Thus, $|rnd_F(t\alpha) - t\alpha| \leq 0.5fmin_D$. That is $t\alpha - 0.5fmin_D \leq rnd_F(t\alpha) \leq$

$t\alpha + 0.5fmin_D$. Therefore, $t - rnd_F(t\alpha) \geq t(1 - \alpha) - 0.5fmin_D$. Inequality $X > 0$ in the clause body implies that $t \geq fmin_D$. Thus, $t - rnd_F(t\alpha) \geq fmin_D(0.5 - \alpha)$. In order for the right hand side to be used as ε , it should be positive, that is $\alpha < 0.5$ should hold.

– $|t\alpha| \geq fmin_N$. Then, $e_F(t\alpha) = \lfloor \log_r |t\alpha| \rfloor + 1$. Thus,

$$|rnd_F(t\alpha) - t\alpha| \leq 0.5r^{\lfloor \log_r |t\alpha| \rfloor + 1 - p} \leq 0.5t\alpha r^{1-p}.$$

Thus, $t - rnd_F(t\alpha) \geq t(1 - \alpha - 0.5\alpha r^{1-p})$. Since $r \geq 2$ and $p \geq 2$, $r^{1-p} \leq 0.5$ and $t - rnd_F(t\alpha) \geq t(1 - 1.25\alpha)$. Since $|t\alpha| \geq fmin_N$, $t > 0$ and $\alpha > 0$ also $t \geq fmin_N/\alpha$ should hold. Then, $t - rnd_F(t\alpha) \geq fmin_N(1 - 1.25\alpha)/\alpha$. To make the right-hand side expression positive $\alpha < 0.8$ should hold. Let $fmin_N(1 - 1.25\alpha)/\alpha$ be a candidate for ε .

Summarising the two cases we obtain that $\alpha < 0.5$ should hold and that ε can be chosen as $\min(fmin_D(0.5 - \alpha), fmin_N(1 - 1.25\alpha)/\alpha)$. Thus,

$$|t|_2 = \left\lfloor \frac{\#}{\min(fmin_D(0.5 - \alpha), fmin_N(1 - 1.25\alpha)/\alpha)} \right\rfloor \text{ proves termination.}$$

It should be noted that in actual Prolog implementations termination of $p(t)$ with respect to the program above may also be observed if $\alpha = 0.5$. However, ISO standard for language arithmetics does not specify the behaviour of $rnd_F(x)$ if x is exactly halfway between two values and rounding upwards in this case will result in non-termination.

Assume now that rnd_F has the *round toward zero* property. In this case $|rnd_F(x)| \leq |x|$ should hold for all x . Since $\alpha > 0$ and $t > 0$, $t\alpha > 0$ and $rnd_F(t\alpha) \geq rnd_F(0)$.

By definition of F^* , $0 \in F^*$, that is $rnd_F(0) = 0$. Thus, $rnd_F(t\alpha) \geq 0$, $rnd_F(t\alpha) \leq t\alpha$ and $t - rnd_F(t\alpha) \geq t(1 - \alpha)$. As above, the body constraint $X > 0$ means $t \geq fmin_D$. Therefore, $t - rnd_F(t\alpha) \geq fmin_D(1 - \alpha)$. The right hand-side is positive for all $0 < \alpha < 1$ and $\varepsilon = fmin_D(1 - \alpha)$. Thus, if rounding happens toward zero, $p(t)$ terminates for all t for all α . \square

Now consider another example, very similar to the one we have seen before, but significantly differing from it in its termination behaviour.

Example 10. Let P be the following program, extending P_3 from Example 1.

$$q(X) \leftarrow X > 0, X1 \text{ is } X - \alpha, q(X1).$$

where $\alpha > 0$. In theory, $q(t)$ should terminate for all α and all t . Choosing $|q(t)| = t$, applying the same technique as above to infer ε and to refine the level-mapping, and assuming rounding to nearest policy and no information loss before rounding, one gets $rnd_F(t - \alpha) \leq (t - \alpha) + 0.5r^{1-p}(t - \alpha)$

if $t - \alpha \geq fmin_N$ and $rnd_F(t - \alpha) \leq (t - \alpha) + 0.5fmin_D$ if $t - \alpha < fmin_N$. Thus, in the first case $t - rnd_F(t - \alpha) \geq \alpha - 0.5r^{1-p}(t - \alpha)$. Since t is not bounded from above, one cannot claim the right hand side expression to be positive for all t , thus, we cannot find ε as above and suspect non-termination. Indeed, for t much bigger than α , $rnd_F(t - \alpha)$ might be equal to t , starting non-terminating execution. For example, if $\alpha = 0.1$, $10^{16} - \alpha$ is computed by the *iso*-mode of SICStus Prolog [22] to be 10^{16} .

Precision of this analysis might be improved if information on the call set is available, such as that $Call(P, q(t_0)) \subseteq \{q(t) \mid t \leq t_0\}$ for a given goal $q(t_0)$. This would provide the desired upper bound on the values of t and we will be able to claim the right hand side expression to be positive for $\alpha > \frac{t_0 r^{1-p}}{2+r^{1-p}}$. This allows us to conclude termination of $q(t_0)$ for $\alpha > \max(\frac{t_0 r^{1-p}}{2+r^{1-p}}, 0.5fmin_D)$. The specified information about the call set can be deduced by the methods suggested in [16].

Similarly to the previous example, if rounding is done toward zero (and still no information is lost before rounding), termination can be shown for all α and all t by observing that $rnd_F(t - \alpha) \leq t - \alpha$ implies $t - rnd_F(t - \alpha) \geq \alpha$ and $\alpha > 0$ is given.

However, if information is lost before rounding, even assuming rounding toward zero doesn't allow us to prove termination. Indeed, if there exists $k > 0$, such that for all t , such that $q(t)$ is in the call set, $t - \alpha < t - k < t$ and $t - k \in F$, then $add_F^*(t, -\alpha) \leq t - k$. In this case, $t - rnd_F(add_F^*(t, -\alpha)) \geq t - add_F^*(t, -\alpha) \geq t - (t - k) = k$. For some initial calls $q(t_0)$ and some α the existence of k follows from the structure of the call set and the optional requirement $r^{p-1} \geq 10^6$ above. For example, if $t_0 = 1.01$ and $\alpha = 0.1$, then the call set is $\{1.01 - 0.1i \mid 0 \leq i \leq 11\}$ and $k = 0.05$ satisfies the conditions above. However, in general, nothing can guarantee that k as above exists. \square

Finally, consider a more realistic example.

Example 11. A bank has special conditions for deposits between 100 and 1000 euros on savings accounts. Every year it grants 5% and charges 1 euro for the account management. The conditions terminate when the deposit reaches 1000 euros. The following program computes the maximal number of years clients may enjoy the special conditions as a function of their investments.

```

years(Investment, Years) ← Investment > 100,
    Investment < 1000, compute(Investment, 1, Years).
compute(Balance, Y, Years) ← Balance < 1000,

```

$NewBalance$ is $1.05 * Balance - 1$, $Y1$ is $Y + 1$,
 $compute(NewBalance, Y1, Years)$.
 $compute(Balance, Y, Years) \leftarrow Balance \geq 1000, Years$ is Y .

We would like to prove termination of $years(i, y)$, where i is a floating-point number and y is a variable. Transforming the program as explained in [20], obtain:

$years(Investment, Years) \leftarrow Investment > 100$,
 $Investment < 1000, Investment < 1000$,
 $compute^{s1 < 1000}(Investment, 1, Years)$.
 $compute^{s1 < 1000}(Balance, Y, Years) \leftarrow Balance < 1000$,
 $NewBalance$ is $1.05 * Balance - 1$, $Y1$ is $Y + 1$,
 $NewBalance < 1000, compute^{s1 < 1000}(NewBalance, Y1, Years)$.
 $compute^{s1 < 1000}(Balance, Y, Years) \leftarrow Balance < 1000$,
 $NewBalance$ is $1.05 * Balance - 1$, $Y1$ is $Y + 1$,
 $NewBalance \geq 1000, compute^{s1 \geq 1000}(NewBalance, Y1, Years)$.
 $compute^{s1 \geq 1000}(Balance, Y, Years) \leftarrow Balance \geq 1000, Years$ is Y .

As explained above, in order to prove termination it is sufficient to show that there exists $\varepsilon > 0$, such that $| compute^{s1 < 1000}(balance, y, years) | - | compute^{s1 < 1000}(new_balance, y1, years) | \geq \varepsilon$, for all calls to $compute^{s1 < 1000}$. The only inequality we have in the second clause is $Balance < 1000$, therefore, similarly to [20], we base the level mapping on $1000 - Balance$, that is, we define $| compute^{s1 < 1000}(balance, y, years) |$ as $1000 - balance$ if $balance < 1000$ and as 0, otherwise. Then, in order to prove termination there should exist $\varepsilon > 0$, such that for all $100 < balance < 1000$

$$(1000 - balance) - (1000 - rnd_F(rnd_F(1.05balance) - 1)) \geq \varepsilon.$$

Assume that the rounding function has the *round to nearest* property. We are going to estimate $rnd_F(rnd_F(1.05balance) - 1)$. Since $balance > 100$, it holds that $1.05balance \geq fmin_N$ and $rnd_F(1.05balance) - 1 \geq fmin_N$. Thus, $rnd_F(rnd_F(1.05balance) - 1) \geq (1.05balance(1 - 0.5r^{1-p}) - 1)(1 - 0.5r^{1-p})$. Then, $rnd_F(rnd_F(1.05balance) - 1) - balance \geq balance(0.05 + 0.2625r^{2(1-p)} - 1.05r^{1-p}) - 1 + 0.5r^{1-p}$. In order the right-hand side expression to be used as ε , it should be positive. First of all, $r^{1-p} \leq 10^{-6}$ implies $0.2625r^{2(1-p)} - 1.05r^{1-p} + 0.05 > 0$. Second,

$$balance > \frac{1 - 0.5r^{1-p}}{0.05 + 0.2625r^{2(1-p)} - 1.05r^{1-p}} \quad (5)$$

should hold. One can show that the fraction on the right-hand side is monotonically increasing as a function of r^{1-p} . Since $r^{1-p} \leq 10^{-6}$ should hold, if $balance > \frac{1-0.5*10^{-6}}{0.05+0.2625*10^{-12}-1.05*10^{-6}}$, then (5) holds for this $balance$ and all possible values of r and p . Inequality $balance > 100$ implies the later condition, and thus, (5). Therefore, $balance(0.05 + 0.2625r^{2(1-p)} - 1.05r^{1-p}) - 1 + 0.5r^{1-p} > 0$ and choosing ε as $balance(0.05+0.2625r^{2(1-p)} - 1.05r^{1-p}) - 1 + 0.5r^{1-p}$ proves termination. \square

5 Toward the automation of the approach

In the previous section we have analysed a number of examples. The analysis followed the same pattern: infer the set of calls [16] (sometimes omitted in the examples, if the decrease can be shown for *all* values), apply the technique of [20] to infer level-mappings, construct acceptability decreases and try to prove them using the formulae recited in Section 3.

The most intriguing step of proving termination is the last one, namely, proving that there exists ε as required. Assume that the rounding is done to the nearest value. In this case Condition 4 implies that for all $x \in \mathcal{R}$ holds, that $lo(x) \leq rnd_F(x) \leq hi(x)$, where lo and hi are defined as

$$lo(x) = \begin{cases} x - 0.5fmin_D & \text{if } |x| < fmin_N \\ x(1 - 0.5r^{1-p}) & \text{if } |x| \geq fmin_N \end{cases}$$

$$hi(x) = \begin{cases} x + 0.5fmin_D & \text{if } |x| < fmin_N \\ x(1 + 0.5r^{1-p}) & \text{if } |x| \geq fmin_N \end{cases}$$

If the rounding is done toward zero, similar definitions for lo and hi can be obtained from Conditions 2 and 3.

Based on the basic inequalities of the form $lo(x) \leq rnd_F(x) \leq hi(x)$ and using well-known principles of interval arithmetics, such as $a \leq x \leq b, c \leq y \leq d \Rightarrow a+c \leq x+y \leq b+d$, allows us to estimate sizes of the call to the head and of the call to the body. For the further analysis we take an underestimate for the size of the call to the head and an overestimate for size of the call to the recursive subgoal.

Example 12. Recall Example 11. We would like to estimate the value of $rnd_F(rnd_F(1.05t) - 1)$. Then, the following holds:

$$\begin{aligned} lo(1.05t) &\leq rnd_F(1.05t) \leq hi(1.05t) \\ rnd_F(lo(1.05t) - 1) &\leq rnd_F(rnd_F(1.05t) - 1) \leq rnd_F(hi(1.05t) - 1) \\ lo(lo(1.05t) - 1) &\leq rnd_F(rnd_F(1.05t) - 1) \leq hi(hi(1.05t) - 1) \quad \square \end{aligned}$$

Next, based on the estimations we distinguish between the cases according to definitions of *lo* and *hi*. Inconsistent cases are rejected and the functions are replaced by their definitions.

Example 13. Consider the underestimate obtained in Example 12. The following cases are distinguished:

- $|1.05t| < fmin_N$ and $|lo(1.05t) - 1| < fmin_N$.
- $|1.05t| < fmin_N$ and $|lo(1.05t) - 1| \geq fmin_N$.
- $|1.05t| \geq fmin_N$ and $|lo(1.05t) - 1| < fmin_N$.
- $|1.05t| \geq fmin_N$ and $|lo(1.05t) - 1| \geq fmin_N$.

In theory, all four cases should be considered. However, recall that t stands for the balance, which in our example is greater than 100, making the first two cases impossible, since $fmin_N < 1$. The definition of *lo* implies the impossibility of the third case as well. The remaining case results in the following underestimate function: $(1.05t(1 - 0.5r^{1-p}) - 1)(1 - 0.5r^{1-p})$. \square

At the next step we construct the difference between the underestimate of the size of the call to the head and the overestimate of the size of the call to the recursive subgoal and try to prove that there exists ε , such that this difference will always be not less than it. If for each clause and each pair—call to the head, call to the recursive subgoal—there exists such a ε , we can take the smallest of them to satisfy the decrease conditions for the whole program (see also Example 9).

Proving the existence of such an ε , can be done by modern computing packages such as Maple [23].

Example 14. Example 13, continued. We have to show that there exists $\varepsilon > 0$, such that for all $t > 100$ and for all r and p , such that $0 < r^{1-p} < 10^{-6}$, $t(0.05 + 0.2625r^{2(1-p)} - 1.05r^{1-p}) - 1 + 0.5r^{1-p} > \varepsilon$. Denoting r^{1-p} by x and running the following command in Maple gives the answer:

```
> minimize(t*(0.05+0.2625*x^2-1.05*x)-1+\
> 0.5*x, t=100..infinity, x=0..0.000001);
3.999895500
```

Since the infimum computed is greater than 0, it can be used as ε . This completes the termination proof. \square

6 Conclusion

We have presented an approach to verification of termination for logic programs with computations depending on real numbers. To the best

of our knowledge this is the first work in this domain. Termination of numerical computations was studied by a number of authors [5, 6, 13, 20].

In [5] it is claimed that an unchanged acceptability condition can be applied to programs in pure Prolog with arithmetic by defining the level mappings on ground atoms with the arithmetic relation to be zero. This approach ignores the actual computation, and thus, its applicability is restricted to programs using some arithmetic but not really relying on them, such as *quicksort*. Dershowitz *et al.* [13] and Serebrenik and De Schreye [20] restricted their attention to integer computations only.

Apt *et al.* [6] provided a declarative semantics, so called Θ -semantics, for Prolog programs with first-order built-in predicates, including arithmetic operations. In this framework the property of strong termination, i.e., finiteness of all LD-trees for all possible goals, was completely characterised based on appropriately tuned notion of acceptability. However, this approach studied termination with respect to ideal real numbers and not with respect to actual floating point computations. Example 1 illustrates that these two notions of termination are orthogonal. We also believe that in order for a termination analyser to be of practical use termination with respect to floating point numbers should be considered.

More research has been done on termination analysis for constraint logic programs [9, 18, 19]. Since numerical computations in Prolog should be written in a way that allows a system to verify their satisfiability we can see numerical computations of Prolog as an *ideal constraint system*. Thus, all the results obtained for ideal constraints systems can be applied. Unfortunately, the research usually restricted to domains isomorphic to \mathcal{N} [18], such as trees and terms. Ruggieri [19] provides theoretical characterisation of termination, but doesn't suggest a method to infer level-mappings required, nor does he study the influence of imprecision inherent to floating-point computations.

References

1. ISO/IEC 10967-1:1994. *Information technology—Language independent arithmetic—Part 1: Integer and floating point arithmetic*. ISO/IEC, 1994.
2. ISO/IEC 10967-2:2001. *Information technology—Language independent arithmetic—Part 2: Elementary numerical functions*. ISO/IEC, 2001.
3. ISO/IEC 13211-1:1995. *Information technology—Programming languages—Prolog—Part 1: General core*. ISO/IEC, 1995.
4. Abderrahmane Aggoun, David Chan, Pierre Dufresne, Eamon Falvey, Hugh Grant, Warwick Harvey, Alexander Herold, Macartney Geoffrey, Micha Meier, David Miller, Shyam Mudambi, Stefano Novello, Bruno Perez, Emmanuel van Rossum, Joachim Schimpf, Kish Shen, Periklis Andreas Tsahageas, and Dominique Henry de Villeneuve. *ECLⁱPS^e User Manual. Release 5.3*, 2001.

5. Krzysztof R. Apt. *From Logic Programming to Prolog*. Prentice-Hall Int. Series in Computer Science. Prentice Hall, 1997.
6. Krzysztof R. Apt, Elena Marchiori, and Catuscia Palamidessi. A declarative approach for first-order built-in's in Prolog. *Applicable Algebra in Engineering, Communication and Computation*, 5(3/4):159–191, 1994.
7. Mike Brady. Open Prolog. Available at <http://www.cs.tcd.ie/open-prolog/>, 2001.
8. Francisco Bueno, Daniel Cabeza Gras, Manuel Carro, Manuel V. Hermenegildo, Pedro López-García, and Germán Puebla. The Ciao Prolog system. Reference manual. Technical Report CLIP3/97.1, School of Computer Science, Technical University of Madrid (UPM), August 1997. Available from <http://www.clip.dia.fi.upm.es/>.
9. Livio Colussi, Elena Marchiori, and Massimo Marchiori. On termination of constraint logic programs. In Ugo Montanari and Francesca Rossi, editors, *Principles and Practice of Constraint Programming - CP'95*, pages 431–448. Springer Verlag, 1995. LNCS 976.
10. Danny De Schreye, Kristof Verschaetse, and Maurice Bruynooghe. A framework for analyzing the termination of definite logic programs with respect to call patterns. In ICOT Staff, editor, *Proc. of the Int. Conf. on Fifth Generation Computer Systems.*, pages 481–488. IOS Press, 1992.
11. Stefaan Decorte and Danny De Schreye. Termination analysis: some practical properties of the norm and level mapping space. In Joxan Jaffar, editor, *Proc. of the 1998 Joint Int. Conf. and Symp. on Logic Programming*, pages 235–249. MIT Press, June 1998.
12. Stefaan Decorte, Danny De Schreye, and Henk Vandecasteele. Constraint-based termination analysis of logic programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 21(6):1137–1195, November 1999.
13. Nachum Dershowitz, Naomi Lindenstrauss, Yehoshua Sagiv, and Alexander Serebrenik. A general framework for automatic termination analysis of logic programs. *Applicable Algebra in Engineering, Communication and Computing*, 12(1-2):117–156, 2001.
14. IEEE Standards Committee 754. *IEEE Standard for binary floating-point arithmetic, ANSI/IEEE Standard 754-1985*. Institute of Electrical and Electronics Engineers, New York, 1985. Reprinted in SIGPLAN Notices, 22(2):9-25, 1987.
15. IT Masters. MasterProLog Programming Environment. Available at <http://www.itmasters.com/>, 2000.
16. Gerda Janssens, Maurice Bruynooghe, and Vincent Englebort. Abstracting numerical values in CLP(H, N). In Manuel V. Hermenegildo and Jan Penjam, editors, *Programming Language Implementation and Logic Programming, 6th International Symposium, PLILP'94*, pages 400–414. Springer Verlag, 1994. LNCS 844.
17. William Kahan. Lecture notes on the status of IEEE standard 754 for binary floating-point arithmetic. Available at <http://grouper.ieee.org/groups/754/>, May 1996.
18. Fred Mesnard. Inferring left-terminating classes of queries for constraint logic programs. In Michael Maher, editor, *Proc. JICSLP'96*, pages 7–21. The MIT Press, 1996.
19. Salvatore Ruggieri. *Verification and validation of logic programs*. PhD thesis, Università di Pisa, 1999.
20. Alexander Serebrenik and Danny De Schreye. Inference of termination conditions for numerical loops in Prolog. In R. Nieuwenhuis and A. Voronkov, editors, *Logic*

- for Programming, Artificial Intelligence, and Reasoning, 8th International Conference, Proceedings*, volume 2250 of *Lecture Notes in Computer Science*, pages 654–668. Springer Verlag, 2001.
21. Alexander Serebrenik and Danny De Schreye. Non-transformational termination analysis of logic programs, based on general term-orderings. In Kung-Kiu Lau, editor, *Logic Based Program Synthesis and Transformation 10th International Workshop, Selected Papers*, volume 2042 of *Lecture Notes in Computer Science*, pages 69–85. Springer Verlag, 2001.
 22. SICS. *SICStus User Manual. Version 3.9*. Swedish Institute of Computer Science, 2002.
 23. Inc. Waterloo Maple. Maple 7, 2001. Consult <http://www.maplesoft.com/>.
 24. Will Winsborough. Multiple specialization using minimal-function graph semantics. *Journal of Logic Programming*, 13(2/3):259–290, 1992.