TEMA Tend. Mat. Apl. Comput., **7**, No. 2 (2006), 249-258.
© *Uma Publicação da Sociedade Brasileira de Matemática Aplicada e Computacional.*

# Interval Term Rewriting System: Toward A Formal Model for Interval Computation

A.X. CARVALHO[1], R.H.N. SANTIAGO[2], Departamento de Informática e Matemática Aplicada – DIMAp, Universidade Federal do Rio Grande do Norte, 59078-970 Natal, RN, Brasil.

**Abstract**. We present a term rewriting system for interval arithmetic (addition, subtraction and multiplication), toward a mathematical model for interval computation. We start presenting a term rewriting system whose rules (directed equations) perform binary floating-point arithmetic, which is based on IEEE-754 Standard. Next, this primitive system is extended with rules for interval arithmetic. Finally, correctness and termination of our system are both discussed.

## 1. Introduction

Computational representation of real numbers on actual machines is usually done by floating-points, which belong to a finite subset $F$ of rational numbers $\mathbb{Q}$ and are used as approximations for real numbers. The abstract definition of such systems is commonly done as a mathematical structure (a model) of the form $\mathbb{F} = \langle F, b, t, e_1, e_2, O\rangle$, where $F \subset \mathbb{Q}$ is the set of all machine numbers, $b$ is the numerical base (binary, decimal etc), $t$ is the precision, $e_1$ the minimum exponent and $e_2$ the maximum exponent of the system, and, finally, $O : \mathbb{R} \to F$, a rounding function. A complete overview about floating-points as a partial algebra can be found at [4].

Since the representation of real numbers as floating-points leads to accumulative rounding errors and lack of error control in successive arithmetic operations, interval computation (and others self-validated methods) arose as a more accurate approach, where a real number is approximated by an interval which contains it. A good introduction for interval analysis and computation can be found at [8].

In this paper, we briefly define a formal model for interval arithmetic, which complete description is in reference [3]. The mathematical computational model is a term rewriting system (TRS) [1, 10], and it is intended to be correct, total, closed, optimal and efficient, in order to provide an operational semantics for languages and systems which make use of intervals as strategy for representation of real numbers. Furthermore, since a TRS is a Post system (see [2]), we also hope to bring up a basis for the development of a model of computability for real numbers.

[1]adrianoxc@gmail.com
[2]regivan@dimap.ufrn.br,regivan.santiago@gmail.com

Section 2. introduces basic concepts and mathematical issues in rewriting systems area. Section 3. presents FTRS, a TRS whose rules perform binary floating-point arithmetic according to IEEE-754 Standard [7]. In Section 4., this primitive TRS is extended with rules for interval arithmetic. Finally, in Section 5., we talk about correctness and termination of our system and we remark some future work.

## 2. Term Rewriting Systems

In order to mechanize as much as possible the equational reasoning, it should be considered the problem of finding a decision procedure for equational theories. This problem is usually instanced in the group theory (see [1] and [10]), defined by the following axioms, where "$*$" is a binary associative operation, "$e$" is the identidy element of that operation, and $i(x)$ designates the inverse of $x$:

$$\begin{aligned} x * e &= x \\ x * i(x) &= e \\ (x * y) * z &= x * (y * z) \end{aligned}$$

The proof that "$e$" is also left-identidy can be done by equational replacement as follows (from [10] pp. 83):

$$\begin{aligned} e * x &= e * (x * e) = e * (x * (i(x) * i(i(x)))) \\ &= e * ((x * i(x)) * i(i(x))) = e * (e * i(i(x))) \\ &= (e * e) * i(i(x)) = e * i(i(x)) = (x * i(x)) * i(i(x)) \\ &= x * (i(x) * i(i(x)) = x * e = x \end{aligned}$$

Automating this proof needs to guess which is the suitable axiom to be applied at each step, in which direction, and possibly to backtrack. The idea of rewriting is to suppress backtracking, by imposing directionality in the use of equations. This introduces the notion of *rewrite rule*:

**Definition 2.1.** *A* ***rewrite rule*** *is an ordered pair of terms denoted by $l \to r$.*

Contrarily to equalities, one rule is applied by replacing an instance of the left-hand side by the same instance of its right-hand side, but never the converse.

**Definition 2.2.** *A* ***term rewriting system (TRS)*** *is a set of rewrite rules.*

Mathematically, a TRS is a structure $\langle T_\Sigma(X), \to\rangle$, where $T_\Sigma(X)$ is a set of symbolic objects and $\to$ is a binary relation.

**Definition 2.3.** *Given a TRS $R$, a* ***rewriting relation****, denoted by $\to_R$, is the binary relation induced on terms by $R$.*

A notable propriety of a rewriting relation resides on its *termination*:

**Definition 2.4.** *A rewriting relation $\to_R$, on the set of terms, is* ***terminating*** *iff there is no infinity descendent chain $t' \to_R t'' \to_R \ldots$.*

## 3. Floating-point TRS

In this section, Floating-point Term Rewriting System (FTRS) is revisited. For short, we present only constructors and addition, as a sample arithmetic operation. The complete system can be seen at [3].

Operations on digits, integers (lists of digits) and floating-points (concatenated fixed size lists) are indexed by "$\mathbb{D}$", "$\mathbb{Z}$" e "$\mathbb{F}$", respectively. The list constructor ":" and the arithmetic on integers were strongly inspired in [9]. See also [11]. Terms representing floating-points are defined by the constructor ";", which concatenates the digit of signal, the list of exponent and the list of fraction (just as in IEEE-754 Standard, the implicit bit of significand is not represented). Constructors express the formats described in [7] pp. 2-3, which is also summarized in [6] pp. 6.

For readability reasons, floating-point constructors stand for precision of three bits for exponent and five bits for fraction, but FTRS can be easily adapted for each precision defined in IEEE-754 Standard, since equations for arithmetic were defined for generic precision. Moreover, we use list variables "$e\_any$" and "$f\_any$" as short for arbitrary exponents "$((END : e_0) : e_1) : e_2$" and arbitrary fractions "$((((END : f_0) : f_1) : f_2) : f_3) : f_4$", respectively, where $e_i$ and $f_i$ are digit variables (e.g. see some rules at section 3.1.).

Constants in Table 1 represent special integers (lists) which must be set according to the desired precision, in this case, three bits for exponent and five for fraction, as mentioned above. Due to the IEEE-754 biased exponent representation, the special exponent "1" (which is interpreted as $1 - bias$) is semantically equivalent to the denormalized exponent "0" (which is interpreted as $-bias + 1$).

| Name | Value | Description |
|---|---|---|
| $e\_size$ | $(END : 1) : 1$ | Bits for exponent. |
| $f\_size$ | $((END : 1) : 0) : 1$ | Bits for fraction. |
| $e\_min$ | $((END : 0) : 0) : 0$ | Minimum exponent. |
| $e\_max$ | $((END : 1) : 1) : 1$ | Maximum exponent. |
| $e\_esp$ | $((END : 0) : 0) : 1$ | Special exponent. |
| $f\_min$ | $((((END : 0) : 0) : 0) : 0) : 0$ | Minimum fraction. |
| $f\_max$ | $((((END : 1) : 1) : 1) : 1) : 1$ | Maximum fraction. |

Table 1: Precision dependent constants

In the following, expertise on IEEE-754 Standard will be assumed.

### 3.1. Constructors

Firstly, we present interpretation of constructors. On later, rules for arithmetic.

$$\text{/* Digits */}$$
$$\|0\| = 0 \quad \|1\| = 1 \quad \|10\| = 2$$

$$\text{/* Positive integer */} \quad \text{/* Negative integer */} \quad \text{/* End of list */}$$
$$\|l : d\| = \|d\| + 2 \times \|l\| \quad \|MINUS(l)\| = -\|l\| \quad \|END\| = 0$$

Let be $s \in \{0,1\}$ and the list variables $e\_any = ((END : e_0) : e_1) : e_2$ and $f\_any = ((((END : f_0) : f_1) : f_2) : f_3) : f_4$, which sizes reflect the precision, arbitrarily chosen, of the system. Floating-points constructors are given by:

$$
\begin{gathered}
\text{/* Zero */} \\
\|s; e\_min; f\_min\| = 0 \\
\text{/* Denormalized number */} \\
\|s; e\_min; f\_any\| = (-1)^{\|s\|} \times (0 + \|f\_any\| \times 2^{-f\_size}) \times 2^{-bias+1}, \\
\text{if } (f\_any \neq f\_min) \\
\text{/* Normalized number */} \\
\|s; e\_any; f\_any\| = (-1)^{\|s\|} \times (1 + \|f\_any\| \times 2^{-f\_size}) \times 2^{\|e\_any\|-bias}, \\
\text{if } (f\_any \neq f\_min) \text{ and } (f\_any \neq f\_max) \\
\text{/* Infinity */} \\
\|s; e\_max; f\_min\| = (-1)^{\|s\|} \times \infty \\
\text{/* Signalling NaN: ``invalid'' */} \\
\|s; e\_max; f\_any\| = SNaN, \\
\text{if } (f_0 = 0) \text{ and } (f\_any \neq f\_min) \\
\text{/* Quiet NaN: ``indeterminate'' */} \\
\|s; e\_max; f\_any\| = QNaN \\
\text{if } (f_0 = 1)
\end{gathered}
$$

## 3.2. Addition

Here addition of floating-points is shown. For underlying operations on digits and integers, see [3, 9].

Some problems arise when we formalize arithmetic on IEEE-754 floats. The first one is how to syntactically represent the implicit digit, which is not captured by constructors, although it is required for operations. The system provides the rules *put* and *cut* to solve this problem.

$$
\begin{gathered}
\text{/* Includes the implicit bit */} \\
put(END) \rightarrow END : 1 \\
put(x : y) \rightarrow put(x) : y \\
\|put(t)\| = \|t\| + \|1\| \times 2^{\|length(t)\|} \\
\text{/* Excludes the implicit bit */} \\
cut(END : y) \rightarrow END \\
cut((w : x) : y) \rightarrow cut(w : x) : y \\
\|cut(t)\| = \|t\| - \|1\| \times 2^{\|length(t)\|-1}
\end{gathered}
$$

Performing addition of floats, we have to equal the exponents of operands. The $rshift$ function increments the least exponent up to the greatest one.

$$
\begin{gathered}
\text{/* Increments the exponent:} \\
\text{fraction is shifted, by truncating right bits (rounding to zero!) */} \\
rshift(x, e\_min) \rightarrow x \\
rshift(x : y, z) \rightarrow rshift(x, z -_{\mathbb{Z}} (END : 1)) \\
\|rshift(t, t')\| = \|t\| \times 2^{1-\|t'\|}
\end{gathered}
$$

Two more auxiliar functions:

/* Switches the signal */
$$symm(0;x;y) \rightarrow 1;x;y$$
$$symm(1;x;y) \rightarrow 0;x;y$$

/* Calculates the size */
$$length(END) \rightarrow END:0$$
$$length(x:y) \rightarrow (END:1) +_{\mathbb{Z}} length(x)$$
$$length(MINUS(x)) \rightarrow length(x)$$

During computations, machines need to map floating-point arithmetic statements into objects which is possibly not IEEE-754 Standard floating-points. The Standard establishes that this objects (called "destinations") must be formatted according to the selected precision.

The above requirement is performed by the $normadd_{\mathbb{F}}$ function, which is defined in four cases, depending on both the exponent value and the resulting fraction size:

/* ...normalized exponent */
$$normadd_{\mathbb{F}}(s;x;y) \rightarrow s; x +_{\mathbb{Z}} (length(y) -_{\mathbb{Z}} f\_size); rshift(y, length(y) -_{\mathbb{Z}} f\_size),$$
$$\text{if } (x \neq e\_min) \text{ and } ((x +_{\mathbb{Z}} (length(y) -_{\mathbb{Z}} f\_size)) \leq e\_max)$$
/* ...denormalized exponent, suitable fraction: nothing to do */
$$normadd_{\mathbb{F}}(s;e\_min;y) \rightarrow s;e\_min;y,$$
$$\text{if } (length(y) = f\_size)$$
/* ...denormalized exponent, too large fraction: normalization required */
$$normadd_{\mathbb{F}}(s;e\_min;y) \rightarrow s;e\_esp;cut(y),$$
$$\text{if } (length(y) > f\_size)$$
/* ...overflow: raises QNaN */
$$normadd_{\mathbb{F}}(s;x;y) \rightarrow s;e\_max;f\_max,$$
$$\text{if } (x +_{\mathbb{Z}} (length(y) -_{\mathbb{Z}} f\_size)) > e\_max$$

The following rules describe addition of floating-points. Firstly, the trivial cases are listed:

/* Pos Zero + Any Pos = Any Pos */
$$0;e\_min;f\_min +_{\mathbb{F}} 0;e'\_any;f'\_any \rightarrow 0;e'\_any;f'\_any$$
/* Pos NaN + Any Pos = Pos NaN */
$$0;e\_max;f\_any +_{\mathbb{F}} 0;e'\_any;f'\_any \rightarrow 0;e\_max;f\_any,$$
$$\text{if } (f\_any \neq f\_min)$$
/* Pos Infinity + Pos Num = Pos Infinity */
$$0;e\_max;f\_min +_{\mathbb{F}} 0;e'\_any;f'\_any \rightarrow 0;e\_max;f\_min,$$
$$\text{if } (e'\_any \neq e\_max)$$
/* Pos Infinity + Pos Infinity = Pos Infinity */
$$0;e\_max;f\_min +_{\mathbb{F}} 0;e\_max;f\_min) \rightarrow 0;e\_max;f\_min$$
/* Pos Infinity + Pos NaN = Pos NaN */
$$0;e\_max;f\_min +_{\mathbb{F}} 0;e\_max;f'\_any \rightarrow 0;e\_max;f'\_any,$$
$$\text{if } (f'\_any \neq f\_min)$$
/* Pos Num + Zero Pos = Pos Num */

$$
\begin{gathered}
0; e\_any; f\_any +_{\mathbb{F}} 0; e\_min; f\_min \rightarrow 0; e\_any; f\_any, \\
\text{if } (e\_any \neq e\_max) \\
\text{/* Pos Num + Pos Infinity = Pos Infinity */} \\
0; e\_any; f\_any +_{\mathbb{F}} 0; e\_max; f\_min \rightarrow 0; e\_max; f\_min, \\
\text{if } (e\_any \neq e\_max) \\
\text{/* Pos Num + Pos NaN = Pos NaN */} \\
0; e\_any; f\_any +_{\mathbb{F}} 0; e\_max; f'\_any \rightarrow 0; e\_max; f'\_any, \\
\text{if } (e\_any \neq e\_max) \text{ and } (f'\_any \neq f\_min)
\end{gathered}
$$

From this point, non-trivial cases of floating-point addition will be considered. Note that each implicit bit is concatenated to its respective fraction by *put*; the greatest exponent is taken as the exponent of the sum, while the least one is right shifted in a manner that the both become equals.

$$
\begin{gathered}
\text{/* Pos Norm Num + Pos Norm Num: } e \geq e' \text{ */} \\
0; e\_any; f\_any +_{\mathbb{F}} 0; e'\_any; f'\_any \rightarrow \\
normadd_{\mathbb{F}}(0; e\_any; cut(put(f\_any) +_{\mathbb{Z}} rshift(put(f'\_any), e\_any -_{\mathbb{Z}} e'\_any)), \\
\text{if } (e\_any \neq e\_min) \text{ and } (e\_any \neq e\_max) \text{ and } (e'\_any \neq e\_min) \text{ and} \\
(e'\_any \neq e\_max) \text{ and } (e\_any \geq e'\_any) \\
\text{/* Pos Norm Num + Pos Norm Num: } e < e' \text{ */} \\
0; e\_any; f\_any +_{\mathbb{F}} 0; e'\_any; f'\_any \rightarrow \\
normadd_{\mathbb{F}}(0; e'\_any; cut(rshift(put(f'\_any), e'\_any -_{\mathbb{Z}} e\_any) +_{\mathbb{Z}} put(f\_any))), \\
\text{if } (e\_any \neq e\_min) \text{ and } (e\_any \neq e\_max) \text{ and } (e'\_any \neq e\_min) \text{ and} \\
(e'\_any \neq e\_max) \text{ and } (e\_any < e'\_any)
\end{gathered}
$$

Next rules sum up a normalized (exponent *e_any*) and a denormalized (exponent *e_min*) number. The system musts to do a check: If $e\_any = e\_esp$, it does not try to equal the exponents, because *e_esp* is semantically equal to *e_min*, the difference is in the implicit bit ("1" for *e_esp* and "0" for *e_min*).

$$
\begin{gathered}
\text{/* Pos Norm Num + Pos Denorm Num: } e = e\_esp \text{ */} \\
0; e\_any; f\_any +_{\mathbb{F}} 0; e\_min; f'\_any \rightarrow \\
normadd_{\mathbb{F}}(0; e\_any; cut(put(f\_any) +_{\mathbb{Z}} f'\_any))), \\
\text{if } (e\_any \neq e\_min) \text{ and } (e\_any \neq e\_max) \text{ and} \\
(f'\_any \neq f\_min) \text{ and } (e\_any = e\_esp) \\
\text{/* Pos Norm Num + Pos Denorm Num: } e > e\_esp \text{ */} \\
0; e\_any; f\_any +_{\mathbb{F}} 0; e\_min; f'\_any \rightarrow \\
normadd_{\mathbb{F}}(0; e\_any; cut(put(f\_any) +_{\mathbb{Z}} rshift(f'\_any, e\_any -_{\mathbb{Z}} e\_esp)), \\
\text{if } (e\_any \neq e\_min) \text{ and } (e\_any \neq e\_max) \text{ and} \\
(f'\_any \neq f\_min) \text{ and } (e\_any > e\_esp) \\
\text{/* Pos Denorm Num + Pos Norm Num: */} \\
0; e\_min; f\_any +_{\mathbb{F}} 0; e'\_any; f'\_any \rightarrow 0; e'\_any; f'\_any +_{\mathbb{F}} 0; e\_min; f\_any, \\
\text{if } (f\_any \neq f\_min) \text{ and } (e'\_any \neq e\_min) \text{ and } (e'\_any \neq e\_max)
\end{gathered}
$$

To sum up two denormalized floating-points, it is enough to sum up their fractions:

$$
\begin{gathered}
\text{/* Pos Denorm Num + Pos Denorm Num: */} \\
0; e\_min; f\_any +_{\mathbb{F}} 0; e\_min; f'\_any \rightarrow normadd_{\mathbb{F}}(0; e\_min; f\_any +_{\mathbb{Z}} f'\_any), \\
\text{if } (f\_any \neq f\_min) \text{ and } (f'\_any \neq f\_min)
\end{gathered}
$$

In order to avoid redundant rules and cyclic rewrite steps, we map addition involving operands with different signals into the counterpart subtraction:

$$\begin{gathered}
\text{/* Positive + Negative */} \\
0; e\_any; f\_any +_{\mathbb{F}} 1; e'\_any; f'\_any \rightarrow 0; e\_any; f\_any -_{\mathbb{F}} 0; e'\_any; f'\_any \\
\text{/* Negative + Positive */} \\
1; e\_any; f\_any +_{\mathbb{F}} 0; e'\_any; f'\_any \rightarrow 0; e'\_any; f'\_any -_{\mathbb{F}} 0; e\_any; f\_any
\end{gathered}$$

The sum of any two negatives operands is mapped into the sum of the respective symmetric values, and then the signal is switched:

$$\begin{gathered}
\text{/* Negative + Negative */} \\
1; e\_any; f\_any +_{\mathbb{F}} 1; e'\_any; f'\_any \rightarrow symm(0; e\_any; f\_any +_{\mathbb{F}} 0; e'\_any; f'\_any)
\end{gathered}$$

## 3.3. Comparison function

Here we improve FTRS by providing the *cmp* function, which compares two floating-points $a$ and $b$ and returns "0" if $a \leq b$, or "1" if $b \leq a$, or "2" if either $a$ or $b$ is a NaN. Next section, *cmp* will be used to identify the minimum of two given floating-points. It should be noted that, according to IEEE-754 Standard, a NaN is incomparable to any another floating-point. This simplified implementation of *cmp* deliberately introduces critical pairs in the system. See [3] pp. 94 for a full description of *cmp*.

$$\begin{gathered}
\text{/* Negative < Positive */} \\
cmp(1; e\_any; f\_any, 0; e'\_any; f'\_any) \rightarrow 0, \\
\text{/* Positive > Negative */} \\
cmp(0; e\_any; f\_any, 1; e'\_any; f'\_any) \rightarrow 1, \\
\text{/* NaN's are not ordered */} \\
cmp(s; e\_any; f\_any, s'; e'\_any; f'\_any) \rightarrow 2, \\
\text{if } (e\_any = e\_max \text{ and } f\_any \neq e\_min) \text{ or } (e'\_any = e\_max \text{ and } f'\_any \neq e\_min) \\
\text{/* cmp(-a, -b) => cmp(b, a) */} \\
cmp(1; e\_any; f\_any, 1; e'\_any; f'\_any) \rightarrow cmp(0; e'\_any; f'\_any,\ 0; e\_any; f\_any) \\
\text{/* cmp(a, b) => cmp(a-b, b-a) */} \\
cmp(0; e\_any; f\_any, 0; e'\_any; f'\_any) \rightarrow \\
cmp(0; e\_any; f\_any -_{\mathbb{F}} 0; e'\_any; f'\_any,\ 0; e'\_any; f'\_any -_{\mathbb{F}} 0; e\_any; f\_any)
\end{gathered}$$

# 4. Interval TRS

Now we present Interval TRS, which extends FTRS with rules for Moore's interval arithmetic, in order to provide a formal model for interval computation. Equations presented here express algorithms from Hickey [5], which is proved to be correct, total, closed, optimal and efficient. See also [8]. Subtraction and multiplication of floats, which were omitted in Section 3. for lack of space, can be found at [3].

Following Hickey, we define an ***IEEE-standard interval*** as a real interval whose endpoints are represented by IEEE floating-point numbers, and we also require that "$-0$" can only appear as a right endpoint, and "$+0$" can only appear as a left endpoint. Moreover, we define an ***Interval NaN (INaN)*** as a pair of floating-points where either left or right element is a NaN. The definition of INaN assures fault tolerance when underlying operations on endpoints result NaN.

For readability reasons, we use in this section simple variables $a$, $b$, $c$, and $d$ for floating-points at interval endpoints. Furthermore, we will refer to their respective signal bits simply as $a_s$, $b_s$, $c_s$ and $d_s$.

## 4.1. Constructors

Order on endpoints is checked by using the auxiliar function *cmp*, which is also useful for distinguishing between intervals and INaN's.

$$\begin{gathered}\text{/* IEEE-standard interval */}\\ \|\langle a,b\rangle\| = \{x \in [-\infty,+\infty] \mid \|a\| \leq x \leq \|b\|\},\\ \text{if } (\mathit{cmp}(a,b)=0)\end{gathered}$$

$$\begin{gathered}\text{/* Interval NaN (INaN) */}\\ \|\langle a,b\rangle\| = \emptyset,\\ \text{if } (\mathit{cmp}(a,b)=2)\end{gathered}$$

## 4.2. Addition

$$\langle a,b\rangle +_{\mathbb{IF}} \langle c,d\rangle \to \langle a +_{\mathbb{F}} c, b +_{\mathbb{F}} d\rangle$$

## 4.3. Subtraction

$$\langle a,b\rangle -_{\mathbb{IF}} \langle c,d\rangle \to \langle a -_{\mathbb{F}} d, b -_{\mathbb{F}} c\rangle$$

## 4.4. Multiplication

$$\begin{gathered}\text{/* Positive} \times \text{Positive */}\\ \langle a,b\rangle \times_{\mathbb{IF}} \langle c,d\rangle \to \langle a \times_{\mathbb{F}} c, b \times_{\mathbb{F}} d\rangle,\\ \text{if } (a_s=0) \text{ and } (b_s=0) \text{ and } (c_s=0) \text{ and } (d_s=0)\\ \text{/* Mixed} \times \text{Positive */}\\ \langle a,b\rangle \times_{\mathbb{IF}} \langle c,d\rangle \to \langle a \times_{\mathbb{F}} d, b \times_{\mathbb{F}} d\rangle,\\ \text{if } (a_s=1) \text{ and } (b_s=0) \text{ and } (c_s=0) \text{ and } (d_s=0)\\ \text{/* Negative} \times \text{Positive */}\\ \langle a,b\rangle \times_{\mathbb{IF}} \langle c,d\rangle \to \langle a \times_{\mathbb{F}} d, b \times_{\mathbb{F}} c\rangle,\\ \text{if } (a_s=1) \text{ and } (b_s=1) \text{ and } (c_s=0) \text{ and } (d_s=0)\\ \text{/* Positive} \times \text{Mixed */}\\ \langle a,b\rangle \times_{\mathbb{IF}} \langle c,d\rangle \to \langle b \times_{\mathbb{F}} c, b \times_{\mathbb{F}} d\rangle,\\ \text{if } (a_s=0) \text{ and } (b_s=0) \text{ and } (c_s=1) \text{ and } (d_s=0)\end{gathered}$$

$$\begin{gathered}\text{/* Mixed}\times\text{Mixed: } a \times d < b \times c,\ a \times c < b \times d \text{ */}\\ \langle a,b\rangle \times_{\mathbb{IF}} \langle c,d\rangle \to \langle a \times_{\mathbb{F}} d, b \times_{\mathbb{F}} d\rangle,\\ \text{if } (a_s=1) \text{ and } (b_s=0) \text{ and } (c_s=1) \text{ and } (d_s=0) \text{ and}\\ (\mathit{cmp}(a \times_{\mathbb{F}} d, b \times_{\mathbb{F}} c)=0) \text{ and } (\mathit{cmp}(a \times_{\mathbb{F}} c, b \times_{\mathbb{F}} d)=0)\\ \text{/* Mixed}\times\text{Mixed: } a \times d > b \times c,\ a \times c < b \times d \text{ */}\end{gathered}$$

$$\langle a,b\rangle \times_{\mathbb{IF}} \langle c,d\rangle \rightarrow \langle b \times_{\mathbb{F}} c, b \times_{\mathbb{F}} d\rangle,$$
$$\text{if } (a_s = 1) \text{ and } (b_s = 0) \text{ and } (c_s = 1) \text{ and } (d_s = 0) \text{ and}$$
$$(cmp(a \times_{\mathbb{F}} d, b \times_{\mathbb{F}} c) = 1) \text{ and } (cmp(a \times_{\mathbb{F}} c, b \times_{\mathbb{F}} d) = 0)$$
$$\text{/* Mixed}\times\text{Mixed: } a \times d < b \times c,\ a \times c > b \times d \text{ */}$$
$$\langle a,b\rangle \times_{\mathbb{IF}} \langle c,d\rangle \rightarrow \langle a \times_{\mathbb{F}} d, a \times_{\mathbb{F}} c\rangle,$$
$$\text{if } (a_s = 1) \text{ and } (b_s = 0) \text{ and } (c_s = 1) \text{ and } (d_s = 0) \text{ and}$$
$$(cmp(a \times_{\mathbb{F}} d, b \times_{\mathbb{F}} c) = 0) \text{ and } (cmp(a \times_{\mathbb{F}} c, b \times_{\mathbb{F}} d) = 1)$$
$$\text{/* Mixed}\times\text{Mixed: } a \times d > b \times c,\ a \times c > b \times d \text{ */}$$
$$\langle a,b\rangle \times_{\mathbb{IF}} \langle c,d\rangle \rightarrow \langle b \times_{\mathbb{F}} c, a \times_{\mathbb{F}} c\rangle,$$
$$\text{if } (a_s = 1) \text{ and } (b_s = 0) \text{ and } (c_s = 1) \text{ and } (d_s = 0) \text{ and}$$
$$(cmp(a \times_{\mathbb{F}} d, b \times_{\mathbb{F}} c) = 1) \text{ and } (cmp(a \times_{\mathbb{F}} c, b \times_{\mathbb{F}} d) = 1)$$
$$\text{/* Mixed } \times \text{ Mixed: INaN */}$$
$$\langle a,b\rangle \times_{\mathbb{IF}} \langle c,d\rangle \rightarrow \langle 0; e\_max; f\_max, 0; e\_max; f\_max\rangle,$$
$$\text{if } (a_s = 1) \text{ and } (b_s = 0) \text{ and } (c_s = 1) \text{ and } (d_s = 0) \text{ and}$$
$$((cmp(a \times_{\mathbb{F}} d, b \times_{\mathbb{F}} c) = 2) \text{ or } (cmp(a \times_{\mathbb{F}} c, b \times_{\mathbb{F}} d) = 2))$$
$$\text{/* Negative } \times \text{ Mixed */}$$
$$\langle a,b\rangle \times_{\mathbb{IF}} \langle c,d\rangle \rightarrow \langle a \times_{\mathbb{F}} d, a \times_{\mathbb{F}} c\rangle,$$
$$\text{if } (a_s = 1) \text{ and } (b_s = 1) \text{ and } (c_s = 1) \text{ and } (d_s = 0)$$
$$\text{/* Positive } \times \text{ Negative */}$$
$$\langle a,b\rangle \times_{\mathbb{IF}} \langle c,d\rangle \rightarrow \langle b \times_{\mathbb{F}} c, a \times_{\mathbb{F}} d\rangle,$$
$$\text{if } (a_s = 0) \text{ and } (b_s = 0) \text{ and } (c_s = 1) \text{ and } (d_s = 1)$$
$$\text{/* Mixed } \times \text{ Negative */}$$
$$\langle a,b\rangle \times_{\mathbb{IF}} \langle c,d\rangle \rightarrow \langle b \times_{\mathbb{F}} c, a \times_{\mathbb{F}} c\rangle,$$
$$\text{if } (a_s = 1) \text{ and } (b_s = 0) \text{ and } (c_s = 1) \text{ and } (d_s = 1)$$
$$\text{/* Negative } \times \text{ Negative */}$$
$$\langle a,b\rangle \times_{\mathbb{IF}} \langle c,d\rangle \rightarrow \langle b \times_{\mathbb{F}} d, a \times_{\mathbb{F}} c\rangle,$$
$$\text{if } (a_s = 1) \text{ and } (b_s = 1) \text{ and } (c_s = 1) \text{ and } (d_s = 1)$$
$$\text{/* Zero } \times \text{ Any */}$$
$$\langle a,b\rangle \times_{\mathbb{IF}} \langle c,d\rangle \rightarrow \langle 0; e\_min; f\_min, 0; e\_min; f\_min\rangle,$$
$$\text{if } (a_s = 0) \text{ and } (b_s = 1)$$
$$\text{/* Any } \times \text{ Zero */}$$
$$\langle a,b\rangle \times_{\mathbb{IF}} \langle c,d\rangle \rightarrow \langle 0; e\_min; f\_min, 0; e\_min; f\_min\rangle,$$
$$\text{if } (c_s = 0) \text{ and } (d_s = 1)$$

## 5. Conclusions/Future Work

We have defined a complete term rewriting system for addition, subtraction and multiplication on IEEE-754-standard floating-point intervals. The system does not formalize division, which termination on integers is still an open problem.

Although no formal proof for correctness was provided, some points may be considered: 1) FTRS is correct in the sense that all rewrite rules express true about IEEE-754 floating-points, and all expressions reduce, in a certain way, to floating-points. 2) Correctness of ITRS, in current version, was not reached, because underlying floating-point operations always apply rounding to zero, but this requirement can be provided by improving $rshift$ function, and so employing downward rounding in operations between left endpoints of intervals and upward rounding at right endpoints.

Another desired improvement for ITRS relies on extend intervals (and arithmetic operations) to the more general class of connected subsets of $\mathbb{R}$. Termination proofs of both FTRS and ITRS systems are left as open problem, but a careful analysis of the rules can show the abscence of infinite reductions.

## References

[1] F. Baader, T. Nipkow, "Term Rewriting and All That", Cambridge University Press, 1998.

[2] W.S. Brainerd, L. Landweber, "Theory of Computation", John Wiley & Sons, New York, USA, 1974.

[3] A.X. Carvalho, "Interval Term Rewriting System: Toward a formal model for interval computation", Master thesis, UFRN-DIMAp, Natal, Brazil, 2005.

[4] D. Goldberg, What every computer scientist should know about floating-point arithmetic, in "ACM Computing Surveys" pp. 5-48, 1991.

[5] T. Hickey, Q. Ju, M. van Emden, Interval arithmetic: from principles to implementation, in "Journal of the ACM" pp. 1038-1068, 2001.

[6] S. Hollasch, IEEE standard 754 floating-point numbers, http://steve.hollasch.net/cgindex/coding/ieeefloat.html.

[7] IEEE, "IEEE standard for binary floating-point arithmetic", IEEE Computer Society Press, 1985.

[8] R.B. Kearfott, Interval computations: Introduction, uses, and resources, in "Euromath Bulletin" pp. 95-112, 1996.

[9] R. Kennaway, Complete term rewrite systems for decimal arithmetic and other total recursive functions, "Second International Workshop on Termination", La Bresse, France, 1995.

[10] C. Kirchner, H. Kirchner, "Rewriting Solving Proving", LORIA, INRIA & CNRS, 2001.

[11] H.R. Walters, H. Zantema, Rewrite systems for integer arithmetic, in RTA pp. 324–338, "Centrum voor Wiskunde en Informatica (CWI)", 1995.