

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

Adriano X. Carvalho,

Regivan H. N. Santiago,

Depto de Informática e Matemática Aplicada, CCET, UFRN,  
59078-970, Natal, RN

E-mail: adriano@ppgsc.ufrn.br, regivan@dimap.ufrn.br

## 1 Introduction

Computational representation of real numbers as floating-point lead to accumulative rounding errors and lack of error control in successive arithmetic operations, as we can see in [2]. Interval approach rises as a more accurate alternative, since it treats a given real as not an approximation, but an interval which contains that real. We present a formal model for interval arithmetic, which is intended to be correct, total, closed, optimal and efficient, in order to provide computational semantic for languages/systems which make use of intervals. We start presenting a term rewriting system (TRS) whose rules (directed equations) perform binary floating-point arithmetic (addition, subtraction and multiplication) according to IEEE-754 Standard [1]. Next, this primitive TRS is extended with rules for interval arithmetic, on intervals limited by floating-points. Finally, correctness and termination of our system is both discussed.

## 2 Floating-point TRS

In this section we define a Floating-point Term Rewriting System (FTRS), which will provide semantic for arithmetic operations performed on endpoints of intervals. Operations on digits, integers (lists of digits) and floats (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 [5] (another good approach can be found in [6]). Some equations were done simpler than in Kennaway’s system, since we have not independence of base as a requisite; another notable difference relies in “direction” of construction: we construct lists from right to left side, in order to model lists with leading zeroes.

Terms representing floating-point 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, implicit bit of significand is not represented). For readability reasons, float constructors stand for precision of two bits

for exponent and three 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. Furthermore, we use list variables “ $e\_any$ ” and “ $f\_any$ ” as short for arbitrary exponents “ $(END : e_0) : e_1$ ” and fractions “ $((END : f_0) : f_1) : f_2$ ”, respectively, where  $e_i$  and  $f_i$  are digit variables.

Constants in Table 1 represent especial integers (lists) which must be set according to desired precision, in this case, two bits for exponent and three for fraction, as mentioned above.

Name	Value	Description
$e\_size$	$(END : 1) : 0$	Bits for exponent.
$f\_size$	$(END : 1) : 1$	Bits for fraction.
$e\_min$	$(END : 0) : 0$	Smallest exponent.
$e\_max$	$(END : 1) : 1$	Greatest exponent.
$e\_esp$	$(END : 0) : 1$	Especial exponent <sup>1</sup> .
$f\_min$	$((END : 0) : 0) : 0$	Smallest fraction.
$f\_max$	$((END : 1) : 1) : 1$	Greatest fraction.

Table 1: Precision dependent constants

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

### 2.1 Constructors

Firstly, we present interpretation of constructors. On later, rules for arithmetic operations are provided. Trivial operations involving Zero, NaN or infinity will be omitted for subtraction and multiplication.

#### 2.1.1 Digits

$$\begin{aligned} ||0|| &= 0 \\ ||1|| &= 1 \\ ||10|| &= 2 \end{aligned}$$

#### 2.1.2 Integers

$$\begin{aligned} & /* Positive integer */ \\ ||t : t' || &= ||t' || + 2 \times ||t || \end{aligned}$$

```

/* Negative integer */
||MINUS(t)|| = -||t||

/* End of list */
||END|| = 0

```

### 2.1.3 Floats

```

/* Zero */
||s; e_min; f_min|| = 0

/* Denormalized number */
||s; e_min; f_any|| =
(-1)||s|| × (0 + ||f_any|| × 2-f_size) × 2-bias+1,
if {1} ⊂ {f0, f1, f2}

/* Normalized number */
||s; e_any; f_any|| =
(-1)||s|| × (1 + ||f_any|| × 2-f_size) × 2||e_any||-bias,
if {0, 1} ⊂ {e0, e1}

/* Infinity */
||s; e_max; f_min|| = (-1)||s|| × ∞

/* Signalling NaN: "invalid" */
||s; e_max; (0 : f1) : f2|| = SNaN,
if {1} ⊂ {f1, f2}

/* Quiet NaN: "indeterminate" */
||s; e_max; (1 : f1) : f2|| = QNaN

```

where  $s, e_i, f_i \in \{0, 1\}$ .

## 2.2 Addition

### 2.2.1 Digits

```

/* Addition of digits */
0 +D 0 → 0
0 +D 1 → 1
1 +D 0 → 1
1 +D 1 → 10
||t +D t' || = ||t|| + ||t' ||

```

### 2.2.2 Integers

```

/* Increment of non-negative integers */
inc(END) → END : 1
inc(x : 0) → x : 1
inc(x : 1) → inc(x) : 0
||inc(t)|| = ||t|| + 1

/* Normalization of integers:
removes the improper digit "10" */
normaddZ(x, 0) → x : 0
normaddZ(x, 1) → x : 1
normaddZ(x, 10) → inc(x) : 0
||normaddZ(t, t') || = ||t' || + 2 × ||t ||

/* Addition of integers */
END +Z END → END

```

```

END +Z (x' : y') → x' : y'
END +Z MINUS(y') → MINUS(y')
(x : y) +Z END → x : y
(x : y) +Z (x' : y') → normaddZ(x +Z x', y +D y')
(x : y) +Z MINUS(y') → (x : y) -Z y'
MINUS(x) +Z END → MINUS(x)
MINUS(x) +Z (x' : y') → (x' : y') -Z MINUS(x)
MINUS(x) +Z MINUS(y') → MINUS(x +Z y')
||t +Z t' || = ||t || + ||t' ||

```

### 2.2.3 Floats

```

/* Include implicit bit */
put(END) → END : 1
put(x : y) → put(x) : y
||put(t)|| = ||t || + ||1 || × 2||length(t)||

/* Exclude implicit bit */
cut(END : 0) → END : 0
cut(END : 1) → END
cut(x : y) → cut(x) : y
||cut(t)|| = ||t || - ||1 || × 2||length(t)||-1

/* Increment exponent:
fraction is shifted, by truncating right bits
(downward rounding!) */
rshift(x, e_min) → x
rshift(x : y, z) → rshift(x, z -Z (END : 1))
||rshift(t, t') || = ||t || × 21-||t' ||

/* Auxiliar functions: */

```

```

/* ...switch signal
symm(0; x; y) → 1; x; y
symm(1; x; y) → 0; x; y
||symm(t)|| = -||t ||

```

```

/* ...calculate size */
length(END) → END : 0
length(x : y) → (END : 1) +Z length(x)
length(MINUS(x)) → length(x)

```

```

/* Normalize fraction: */

```

```

/* ...normalized exponent case */
normaddF(s; x; y) → s; x +Z (length(y) -Z
f_size); rshift(y, length(y) -Z f_size),
if (x ≠lex e_min) and
(x +Z (length(y) -Z f_size)) ≤lex e_max

```

```

/* ...denormalized exponent case */
normaddF(s; e_min; y) →
s; e_esp +Z (length(y) -Z f_size); rshift(y, e_esp +Z
(length(y) -Z f_size)),
if length(y) >lex f_size

```

```

/* ...overflow: raises QNaN */
normaddF(s; x; y) → s; e_max; f_max,
if (x +Z (length(y) -Z f_size)) >lex e_max

```

```

/* Addition of floating-points: */

```

```

/* Pos Zero + Any Pos = Any Pos */
0; e_min; f_min +F 0; e'_any; f'_any →
0; e'_any; f'_any
/* Pos NaN + Any Pos = Pos NaN */
0; e_max; f_any +F 0; e'_any; f'_any →
0; e_max; f_any,
if {1} ⊂ {f0, f1, f2}
/* Pos Infinity + Pos Num = Pos Infinity */
0; e_max; f_min +F 0; e'_any; f'_any →
0; e_max; f_min,
if {0} ⊂ {e0, e1}
/* Pos Infinity + Pos Infinity = Pos Infinity */
0; e_max; f_min +F 0; e_max; f_min →
0; e_max; f_min
/* Pos Infinity + Pos NaN = Pos NaN */
0; e_max; f_min +F 0; e_max; f'_any →
0; e_max; f'_any,
if {1} ⊂ {f'0, f'1, f'2}
/* Pos Num + Zero Pos = Pos Num */
0; e_any; f_any +F 0; e_min; f_min →
0; e_any; f_any,
if {0} ⊂ {e0, e1}
/* Pos Num + Pos Infinity = Pos Infinity */
0; e_any; f_any +F 0; e_max; f_min →
0; e_max; f_min,
if {0} ⊂ {e0, e1}
/* Pos Num + Pos NaN = Pos NaN */
0; e_any; f_any +F 0; e_max; f'_any →
0; e_max; f'_any,
if {0} ⊂ {e0, e1} and {1} ⊂ {f'0, f'1, f'2}

/* Pos Norm Num + Pos Norm Num: e ≥ e' */
0; e_any; f_any +F 0; e'_any; f'_any →
normaddF(0; e_any; cut(put(f_any)+Z
rshift(put(f'_any), e_any -Z e'_any))),
if {0, 1} ⊂ {e0, e1} ∩ {e'0, e'1} and e_any ≥lex e'_any

/* Pos Norm Num + Pos Norm Num: e < e' */
0; e_any; f_any +F 0; e'_any; f'_any →
normaddF(0; e'_any; cut(rshift(put(f'_any),
e'_any -Z e_any)+Z put(f_any))),
if {0, 1} ⊂ {e0, e1} ∩ {e'0, e'1} and e_any <lex e'_any

/* Pos Norm Num + Pos Denorm Num: e = e_esp */
0; e_any; f_any +F 0; e_min; f'_any →
normaddF(0; e_any; cut(put(f_any) +Z f'_any))),
if {0, 1} ⊂ {e0, e1}, {1} ⊂ {f'0, f'1, f'2} and
e_any =lex e_esp

/* Pos Norm Num + Pos Denorm Num: e > e_esp */
0; e_any; f_any +F 0; e_min; f'_any →
normaddF(0; e_any; cut(put(f_any)+Z
rshift(f'_any, e_any -Z e_esp))),
if {0, 1} ⊂ {e0, e1} and {1} ⊂ {f'0, f'1, f'2} and
e_any >lex e_esp

/* Pos Denorm Num + Pos Norm Num: */
0; e_min; f_any +F 0; e'_any; f'_any →
0; e'_any; f'_any +F 0; e_min; f_any,

```

if {1} ⊂ {f<sub>0</sub>, f<sub>1</sub>, f<sub>2</sub>} and {0, 1} ⊂ {e'<sub>0</sub>, e'<sub>1</sub>}

```

/* Pos Denorm Num + Pos Denorm Num: */
0; e_min; f_any +F 0; e_min; f'_any →
normaddF(0; e_min; f_any +Z f'_any),
if {1} ⊂ {f0, f1, f2} ∩ {f'0, f'1, f'2}

```

```

/* Positive + Negative */
0; e_any; f_any +F 1; e'_any; f'_any →
0; e_any; f_any -F 0; e'_any; f'_any
/* Negative + Positive */
1; e_any; f_any +F 0; e'_any; f'_any →
0; e'_any; f'_any -F 0; e_any; f_any
/* Negative + Negative */
1; e_any; f_any +F 1; e'_any; f'_any →
symm(0; e_any; f_any +F 0; e'_any; f'_any)

```

## 2.3 Subtraction

### 2.3.1 Digits

```

/* Subtraction of digits */
0 -D 0 → 0
0 -D 1 → MINUS(1)
1 -D 0 → 1
1 -D 1 → 0
||t -D t'|| = ||t|| - ||t'||

```

### 2.3.2 Integers

```

/* Decrement of positive integers */
dec(x : 1) → x : 0
dec(x : 0) → dec(x) : 1
||dec(t)|| = ||t|| - 1

/* Normalization of integers:
removes negative digits from lists */
normsubZ(END, 0) → END
normsubZ(END, 1) → END : 1
normsubZ(END, MINUS(1)) →
MINUS(END : 1)
normsubZ(x : y, 0) → (x : y) : 0
normsubZ(x : y, 1) → (x : y) : 1
normsubZ(x : y, MINUS(1)) → dec(x : y) : 1
normsubZ(MINUS(x), 0) → MINUS(x : 0)
normsubZ(MINUS(x), 1) → MINUS(dec(x) : 1)
normsubZ(MINUS(x), MINUS(1)) →
MINUS(x : 1)
||normsubZ(t, t')|| = ||t'|| + 2 × ||t||

/* Subtraction of integers */
END -Z END → END
END -Z (x : y) → MINUS(x : y)
END -Z MINUS(y') → y'
(x : y) -Z END → (x : y)
(x : y) -Z (x' : y') → normsubZ(x -Z x', y -D y')
(x : y) -Z MINUS(y') → (x : y) +Z y'
MINUS(x) -Z END → MINUS(x)
MINUS(x) -Z (x' : y') → MINUS(x +Z (x' : y'))
MINUS(x) -Z MINUS(y') → y' -Z x
||t +Z t'|| = ||t|| + ||t'||

```

### 2.3.3 Floats

```

/* Decrement exponent: */
/* ...calculate sum of digits */
digsum(END) → 0
digsum(x : 0) → digsum(x)
digsum(x : 1) → 1 + digsum(x)
||digsum(...((d0 : d1) : d2) : ...) : dn)|| =
||d0|| + ||d1|| + ||d2|| + ... + ||dn||

/* ...lshift truncates all left 0's; to do this,
we apply digsum in z to inform hoped 1's. */
lshift(x : y, 0) → END
lshift(x : 0, z) → lshift(x, z) : 0
lshift(x : 1, z) → lshift(x, z -Z (END : 1)) : 1

/* ...fit, in other hand, truncates exactly
the informed count of 0's. */
fit(x : y, f_size) → END
fit(x : y, z) → fit(x, z +Z (END : 1)) : y,
if z <lex f_size

/* Auxiliar functions: */

/* ...leading counts left 0's removed by lshift. */
leading(x : y) → length(x : y) -Z
length(lshift(x : y, digsum(x : y)))

/* ...and rpad pads right side with that 0's. */
rpad(x : y, 0) → x : y
rpad(x : y, z) → pad((x : y) : 0, z -Z (END : 1))

/* Normalize fraction: */

/* ...normalized exponent case */
normsubF(s; w : x; y : z) →
s; (w : x) -Z leading(y : z); rpad(cut(lshift(y : z,
digsum(y : z))), leading(y : z)),
if (w : x) ≠lex e_min and
((w : x) -Z leading(y : z)) ≥lex e_esp

/* ...denormalization of fraction*/
normsubF(s; w : x; y : z) →
s; e_any; rpad(fit(cut(y : z),
(w : x) -Z e_esp), (w : x) -Z e_esp),
if (w : x) ≠lex e_min and
((w : x) -Z leading(y : z)) <lex e_esp

/* ...zero */
normsubF(s; w : x; ((0 : 0) : 0) : 0) →
s; e_any; cut(((0 : 0) : 0) : 0),
if (w : x) ≠lex e_min

/* ...desnormalized exponent case */
normsubF(s; e_any; y : z) → s; e_any; y : z

/* ...negative fraction case */
normsubF(s; w : x; MINUS(z)) →
symm(normsubF(s; w : x; z))

```

```

/* Subtraction of floating-points, excluding trivial
operations involving Zero, NaN, infinity: */

/* Pos Norm Num - Pos Norm Num: e ≥ e' */
0; e_any; f_any -F 0; e'_any; f'_any →
normsubF(0; e_any; put(f_any) -Z
rshift(put(f'_any), e_any -Z e'_any)),
if {0, 1} ⊂ {e0, e1} ∩ {e'0, e'1} and
e_any ≥lex e'_any

/* Pos Norm Num - Pos Norm Num: e < e' */
0; e_any; f_any -F 0; e'_any; f'_any →
normsubF(0; e'_any; rshift(put(f_any),
e'_any -Z e_any) -Z put(f'_any)),
if {0, 1} ⊂ {e0, e1} ∩ {e'0, e'1} and
e_any <lex e'_any

/* Pos Norm Num-Pos Desnorm Num e = e_esp */
0; e_any; f_any -F 0; e_min; f'_any →
normsubF(0; e_any; put(f_any) -Z f'_any),
if {0, 1} ⊂ {e0, e1}, {1} ⊂ {f'0, f'1, f'2} and
e_any =lex e_esp

/* Pos Norm Num-Pos Desnorm Num e > e_esp */
0; e_any; f_any -F 0; e_min; f'_any →
normsubF(0; e_any; put(f_any) -Z
rshift(f'_any, e_any -Z e_esp)),
if {0, 1} ⊂ {e0, e1}, {1} ⊂ {f'0, f'1, f'2} and
e_any >lex e_esp

/* Pos Desnorm Num - Pos Norm Num */
0; e_min; f_any -F 0; e'_any; f'_any →
symm(0; e'_any; f'_any -F 0; e_min; f_any),
if {1} ⊂ {f0, f1, f2} and {0, 1} ⊂ {e'0, e'1}

/* Pos Desnorm Num - Pos Desnorm Num */
0; e_min; f_any -F 0; e_min; f'_any →
normsubF(0; e_min; f_any -Z f'_any),
if {1} ⊂ {f0, f1, f2} ∩ {f'0, f'1, f'2}

/* Positive - Negative */
0; e_any; f_any -F 1; e'_any; f'_any →
0; e_any; f_any +F 0; e'_any; f'_any
/* Negative - Positive */
1; e_any; f_any -F 0; e'_any; f'_any →
symm(0; e'_any; f'_any +F 0; e_any; f_any)
/* Negative - Negative */
1; e_any; f_any -F 1; e'_any; f'_any →
0; e'_any; f'_any -F 0; e_any; f_any

```

## 2.4 Multiplication

### 2.4.1 Digits

```

0 ×D 0 → END
0 ×D 1 → END
1 ×D 0 → END
1 ×D 1 → END : 1
||t ×D t'|| = ||t|| × ||t'||

```

## 2.4.2 Integers

```

/* Multiplication for 2 */
timestwo(END) → END
timestwo(x : y) → (x : y) : 0
timestwo(MINUS(x)) → MINUS(timestwo(x))
||timestwo(t)|| = 2 × ||t||

/* Multiplication of integers */
END ×ℤ END → END
END ×ℤ (x' : y') → END
END ×ℤ MINUS(y') → END
(x : y) ×ℤ END → END
(x : y) ×ℤ (x' : y') → (y ×ℚ y') +ℤ
timestwo(((END : y) ×ℤ x') + (x ×ℤ (END : y'))) +ℤ
timestwo(timestwo(x ×ℤ x'))
(x : y) ×ℤ MINUS(y') → MINUS((x : y) ×ℤ y')
MINUS(x) ×ℤ END → END
MINUS(x) ×ℤ (x' : y') → MINUS(x ×ℤ (x' : y'))
MINUS(x) ×ℤ MINUS(y') → x ×ℤ y'
||t ×ℤ t'|| = ||t|| × ||t'||

```

## 2.4.3 Floats

```

/* Normalization of fraction: */

/* ...normalized exponent */
normmultℝ(s; x; y) →
s; x +ℤ (length(y) -ℤ f_size);
rshift(y, length(y) -ℤ f_size),
if (x ≠lex e_min) and
(x +ℤ (length(y) -ℤ f_size)) ≤lex e_max

/* ...overflow: raises QNaN */
normmultℝ(s; x; y) → s; 1 : 1; (1 : 1) : 1,
if (x +ℤ (length(y) -ℤ f_size)) >lex e_max

```

/\*Multiplication of floating-points, excluding trivial operations involving Zero,NaN,infinity:\*/

```

/* Pos Norm Num × Pos Norm Num: */
0; e_any; f_any ×ℝ 0; e'_any; f'_any →
normmultℝ(0; ((e_any) -ℤ f_size) +ℤ
((e'_any) -ℤ f_size);
cut(put(f_any) ×ℤ put(f'_any))),
if {0, 1} ⊂ {e0, e1} ∩ {e'0, e'1}

```

```

/* Pos Norm Num × Pos Denorm Num: */
0; e_any; f_any ×ℝ 0; e_min; f'_any →
normmultℝ(0; ((e_any) -ℤ f_size) +ℤ
(e_esp -ℤ f_size);,
cut(put(f_any) ×ℤ f'_any)),
if {0, 1} ⊂ {e0, e1}, {1} ⊂ {f'0, f'1, f'2}

```

```

/* Pos Denorm Num × Pos Norm Num: */
0; e_min; f_any ×ℝ 0; e'_any; f'_any →
normmultℝ(0; (e_esp -ℤ f_size) +ℤ
((e'_any) -ℤ f_size);,
cut(f_any ×ℤ put(f'_any))),
if {1} ⊂ {f0, f1, f2} e {0, 1} ⊂ {e'0, e'1}

```

```

/* Pos Denorm Num × Pos Norm Num: */
0; e_min; f_any ×ℝ 0; e_min; f'_any →
normmultℝ(0; (e_esp -ℤ f_size) +ℤ
(e_esp -ℤ f_size);,
cut(f_any ×ℤ f'_any)),
if {1} ⊂ {f0, f1, f2} ∩ {f'0, f'1, f'2}

```

```

/* Positive × Negative */
0; e_any; f_any ×ℝ 1; e'_any; f'_any →
symm(0; e_any; f_any ×ℝ 0; e'_any; f'_any)
/* Negative × Positive */
1; e_any; f_any ×ℝ 0; e'_any; f'_any →
symm(0; e'_any; f'_any ×ℝ 0; e_any; f_any)
/* Negative × Negative */
1; e_any; f_any ×ℝ 1; e'_any; f'_any →
0; e_any; f_any ×ℝ 0; e'_any; f'_any)

```

## 2.5 Comparison function

```

/* Compares two given floats and returns
“0” if f ≤ f', or “1” otherwise */
cmp(s; e_any; f_any, s'; e'_any; f'_any) → 0,
if s = 1 and s' = 0
cmp(s; e_any; f_any, s'; e'_any; f'_any) → 1,
if s = 0 and s' = 1
cmp(s; e_any; f_any, s'; e'_any; f'_any) →
cmp(symm(s'; e'_any; f'_any),
symm(s; e_any; f_any))
if s = 1 and s' = 1
cmp(s; e_any; f_any, s'; e'_any; f'_any) →
cmp(s; e_any; f_any -ℝ s'; e'_any; f'_any,
s'; e'_any; f'_any -ℝ s; e_any; f_any),
if s = 0 and s' = 0

```

## 3 Interval TRS

In this section we extend FTRS with rules for interval arithmetic, in order to show how Interval TRS (ITRS) can be used to provide a formal model for interval computation. Equations presented here express algorithms from Hickey [3], which is proved to be correct, total, closed, optimal and efficient. See also [4].

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. We check order on endpoints using auxiliary function “*cmp*”.

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  $d_s$ . We also abuse language using “0” as short for the term “0;  $e_{min}$ ;  $f_{min}$ ”.

### 3.1 Constructor

$$\| \langle a, b \rangle \| = \{x \in \mathbb{R} \mid \|a\| \leq x \leq \|b\|\},$$

if  $cmp(a, b) = 0$

### 3.2 Addition

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

### 3.3 Subtraction

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

### 3.4 Multiplication

$$\begin{aligned} & /* Positive \times Positive */ \\ & \langle a, b \rangle \times_{\mathbb{IF}} \langle c, d \rangle \rightarrow \langle a \times_{\mathbb{F}} c, b \times_{\mathbb{F}} d \rangle, \\ & \text{if } a_s = 0, b_s = 0, b \neq 0, c_s = 0, d_s = 0 \text{ and } d \neq 0 \end{aligned}$$

$$\begin{aligned} & /* Mixed \times Positive */ \\ & \langle a, b \rangle \times_{\mathbb{IF}} \langle c, d \rangle \rightarrow \langle a \times_{\mathbb{F}} d, b \times_{\mathbb{F}} d \rangle, \\ & \text{if } a_s = 1, b_s = 0, c_s = 0, d_s = 0 \text{ and } d \neq 0 \end{aligned}$$

$$\begin{aligned} & /* Negative \times Positive */ \\ & \langle a, b \rangle \times_{\mathbb{IF}} \langle c, d \rangle \rightarrow \langle a \times_{\mathbb{F}} d, b \times_{\mathbb{F}} c \rangle, \\ & \text{if } a_s = 1, a \neq 0, b_s = 1, c_s = 0, d_s = 0 \text{ and } d \neq 0 \end{aligned}$$

$$\begin{aligned} & /* Positive \times Mixed */ \\ & \langle a, b \rangle \times_{\mathbb{IF}} \langle c, d \rangle \rightarrow \langle b \times_{\mathbb{F}} c, b \times_{\mathbb{F}} d \rangle, \\ & \text{if } a_s = 0, b_s = 0, b \neq 0, c_s = 0 \text{ and } d_s = 1 \end{aligned}$$

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

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

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

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

$$\begin{aligned} & /* Negative \times Mixed */ \\ & \langle a, b \rangle \times_{\mathbb{IF}} \langle c, d \rangle \rightarrow \langle a \times_{\mathbb{F}} d, a \times_{\mathbb{F}} c \rangle, \\ & \text{if } a_s = 1, a \neq 0, c_s = 0 \text{ and } d_s = 1 \end{aligned}$$

$$\begin{aligned} & /* Positive \times Negative */ \\ & \langle a, b \rangle \times_{\mathbb{IF}} \langle c, d \rangle \rightarrow \langle b \times_{\mathbb{F}} c, a \times_{\mathbb{F}} d \rangle, \\ & \text{if } a_s = 0, b_s = 0, b \neq 0, c_s = 1 \text{ and } c \neq 0 \end{aligned}$$

$$\begin{aligned} & /* Mixed \times Negative */ \\ & \langle a, b \rangle \times_{\mathbb{IF}} \langle c, d \rangle \rightarrow \langle b \times_{\mathbb{F}} c, a \times_{\mathbb{F}} c \rangle, \\ & \text{if } a_s = 1, b_s = 0, c_s = 1 \text{ and } c \neq 0 \end{aligned}$$

$$\begin{aligned} & /* Negative \times Negative */ \\ & \langle a, b \rangle \times_{\mathbb{IF}} \langle c, d \rangle \rightarrow \langle b \times_{\mathbb{F}} d, a \times_{\mathbb{F}} c \rangle, \\ & \text{if } a_s = 1, a \neq 0, c_s = 1 \text{ and } c \neq 0 \end{aligned}$$

$$\begin{aligned} & /* Zero \times Any */ \\ & \langle a, b \rangle \times_{\mathbb{IF}} \langle c, d \rangle \rightarrow \\ & (0; e\_min; f\_min, 0; e\_min; f\_min), \\ & \text{if } a = 0 \text{ and } b = 0 \end{aligned}$$

$$\begin{aligned} & /* Any \times Zero */ \\ & \langle a, b \rangle \times_{\mathbb{IF}} \langle c, d \rangle \rightarrow \\ & (0; e\_min; f\_min, 0; e\_min; f\_min), \\ & \text{if } c = 0 \text{ and } d = 0 \end{aligned}$$

## 4 Conclusions/Future Work

We have defined a complete term rewriting system for arithmetic (addition, subtraction and multiplication) on IEEE-754-standard floating-point intervals. Although no formal proof for correctness was presented, 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 to floating-points. 2) Correctness of ITRS, in current version, was not reached, because underlying float operations always apply downward rounding, but this requisite can be provided by improving *rshift* function, and so employing upward rounding in operations between right endpoints of intervals. Another desired improvement for ITRS relies in extend intervals (and arithmetic operations) to the more general class of connected subsets of  $\mathbb{R}$ . Termination of both FTRS and ITRS systems are left as an open problem.

## References

- [1] IEEE Computer Society Standards Committee. IEEE standard for binary floating-point arithmetic, *IEEE Computer Society Press*, 18 (1985).
- [2] D. Goldberg. What every computer scientist should know about floating-point arithmetic, in “ACM Computing Surveys” pp. 5-48, 1991.
- [3] T. Hickey, Q. Ju and M. van Emden. Interval arithmetic: From principles to implementation, in “Journal of the ACM” pp. 1038-1068, 2001.
- [4] R. B. Kearfott. Interval computations: Introduction, uses, and resources, in “Euromath Bulletin” pp. 95-112, 1996.
- [5] R. Kennaway. Complete term rewrite systems for decimal arithmetic and other total recursive functions, “Second International Workshop on Termination”, La Bresse, France, 1995.
- [6] H. R. Walters and H. Zantema. Rewrite systems for integer arithmetic, “Centrum voor Wiskunde en Informatica (CWI)”, 1995.