Comput Syst Sci & Eng (2018) 6: 447–455
© 2018 CRL Publishing Ltd

International Journal of
Computer Systems
Science & Engineering

Reliable Approximated Number System with Exact Bounds and Three-valued Logic

Reeseo Cha[1], **Wonhong Nam**[2,*], **Jin-Young Choi**[1]

[1] *Department of Computer Science and Engineering, Korea university, 145 Anam-ro, Seongbuk-gu, Seoul 02841, Korea*
E-mail: reeseo@korea.ac.kr, choi@formal.korea.ac.kr
[2] *Department of Software, Konkuk university, 120 Neungdong-ro, Gwangjin-gu, Seoul 05029, Korea*

Many programming languages provides mechanism to guarantee the error ranges of exact numbers and intervals. However, when they are integrated with unreliable approximated numbers, we cannot rely on the error-ranges anymore. Such unreliable error-ranges may cause serious errors in programs, and especially in safety critical systems they cost us huge amount of money and/or threaten human's life. Hence, in this paper, we propose a novel number system to safely perform arithmetic operations with guaranteed error ranges. In the number system, exact numbers are separated from approximated numbers, and approximated numbers with strictly guaranteed error-ranges are again separated from unwarranted numbers such as floating-point numbers. A three-valued logic is also shipped with our number system to appropriately deal with uncertainties due to approximations. A prototype implementation of our number system in Python is demonstrated. With this module, we can more reliably execute operations on numbers and make judgments on the conditions involving numbers

Keywords: Formal Methods, Approximated Number System, Exact Bounds, Three-valued Logic

1. INTRODUCTION

A number of modern programming languages and computer algebra systems provide unlimited integers and rational numbers with symbolic computation for *exact arithmetic* [1]. Some of them also support various ways to deal with approximated numbers more precisely, such as arbitrary-precision decimal arithmetic [2]. Moreover, the *interval arithmetic* [3] is a dedicated approximation system where error-ranges are strictly guaranteed during the arithmetic operations. These systems, however, are not so well integrated with unreliable approximations such as IEEE 754 floating-point numbers [4] which are extensively used. For example, the class method `from_float` of the `Fraction` class that is a rational number type in Python [5], maps 0.3 not to $\frac{3}{10}$ but to $\frac{5404319552844595}{18014398509481984}$. Indeed, this is the fractional representation of 0.29999999999999999, which is an erroneous result of approximating intended 0.3 in the IEEE 754 format. Constructions of rational numbers from floating-point numbers using this method let the errors invade from floating-point numbers into rational numbers. The problem may break the reliability of the entire rational numbers in the fraction module of Python. For another example, `(3 ** 0.5) ** 2` in Python is evaluated not to 3 but to 2.9999999999999996, although $(\sqrt{3})^2 = 3$ mathematically.

To confidently rely on number systems, we claim that inexact numbers should be strictly distinguished from exact numbers and intervals, and that the inexactness invades the world of exactness minimally and appropriately. Especially, conditional branches in a program should not be affected inappropriately by *uncertainties* due to the inexactness of the approximated numbers.

Hence, to appropriately deal with uncertainties, we propose a novel number system where exact numbers and intervals are explicitly separated from the inexact approximated numbers and carefully integrated with them, along with a three-valued logic system [6]. With this number system, one can safely per-

*Corresponding author. E-mail: wnam@konkuk.ac.kr

form arithmetic operations with guaranteed error ranges, and can judge conditions without falling into any pitfall.

The rest of this paper is organized as follows. In Section 2, some important recent studies in closely related areas are briefly introduced. In Section 3, we define, as the basic components of our number system, three classes of numbers: exact numbers, proper intervals, and unwarranted numbers, as shown in Table 1. Based on these classes, we define a set of arithmetic operations and coercion rules among these three classes in Section 4. In Section 5, we define logical operations (e.g., equalities and order relations) between them using three-valued logic. We present a prototype implementation in Python in Section 6 and finally we give a conclusion in Section 7.

Table 1 Three classes of numbers

Class	Exactness	Guarantee
exact numbers	exact	location on the number line
proper intervals	approximated	possible range
unwarranted numbers	approximated	none

2. RELATED WORKS

To detour errors resulted from number systems with limited precision such as IEEE 754 arithmetic [4], many efforts have been done so far. These efforts can be roughly classified into several groups according to their goals and directions.

Several studies [1, 7] have aimed for complete liberation from the limits of precision, under the name of exact arithmetic. Typical examples of these include virtually unbounded integers and rational numbers. A number of modern computer algebra systems and programming languages already have these numeric data types built-in.

Symbolic computation have also played important role in this area, guaranteeing that numeric objects can preserve their mathematical semantics without any loss during various arithmetic operations. Mathematica [8] is a typical example of systems utilizing symbolic computation, whereas MATLAB [9] primarily utilizes numerical analysis. The NumPy [10] also provides various algebraic methods to symbolically manipulate numbers and matrices.

Some other studies have aimed for sufficiently fine control of necessary precision. The GNU Multiple Precision (GMP) arithmetic library [11] and MPFR [12] based on it are well known examples of arbitrary precision systems. The Decimal library of Python also provides a number system whose precision can be controlled by programmers exquisitely.

The interval arithmetic [3] is another tool to manipulate approximated numbers within required precision, guaranteeing that quantities of any errors are completely under control even when the error diverges. Many aspects of intervals have been studied, though exact numbers have not been used as the ends of intervals. Three-valued logic [6] itself has also been studied intensively, but it has not been used as a tool to control the semantics of intervals which include "unknown position" on the number line.

To the best of our knowledge, there is no work which combines these techniques together, especially for the sake of formal guarantee where the exactness of numeric operations and logical operations play important role, e.g., cyber-physical systems [13, 14, 15]. Our work makes use of the three-valued logic to ensure the formal correctness of an interval arithmetic system.

3. NUMERIC DATATYPES

In our number system, all the numbers are conceptually categorized into three classes as follows. A number n is:

- an *exact number* if its location on the *number line* can be determined as a point and represented exactly. The class of exact numbers is actually a representable subset of algebraic numbers, i.e., countably infinite set of numbers each element of which has its unique normal form. In our prototype implementation, exact numbers consist of all the rational numbers and some portion of algebraic surd numbers.
- a *proper interval* if its exact location on the number line is unknown but its possible range can be strictly bounded as a line segment and represented exactly. Two distinct ends of a proper interval should be exact numbers, and can be either open or closed.
- an *unwarranted number* if its bounds or possible range of location cannot be guaranteed. Even when we conceptually know its bounds, if we cannot represent them exactly with our exact number, then it also is an unwarranted number. A number in this class contains only blurred, unreliable information about its location on the number line.

Mathematically, the set **RelNum** of all the reliable numbers is a union of three mutually disjoint sets E, I and U, which denote the set of exact numbers, intervals and unwarranted numbers, respectively.

3.1 Exact numbers

From a theoretical point of view, the main goal of our number system is to provide reliability on the arithmetic exactness and logical certainty, and hence we are concerned with rather formal guarantee than usability. For this main goal, any set $E \subset \mathbb{R}$ is sufficient as the set of exact numbers for our number system, provided that:

- E is countably infinite, i.e., $|E| = |\mathbb{N}|$,
- for any $r \in \mathbb{R}$, there exists $a, b \in E$ such that $a \le r \wedge r \le b$, and
- there exists a canonical form in which every element in E can be represented uniquely. Formally, there exists a positive natural number n, and a set $D \subseteq \mathbb{Z}^n$, and a decidable n-ary predicate over $\mathbb{Z}^n$ which determines any n-tuple of natural numbers is an element of D or not, and an n-ary *bijective* constructor function from D to E.

For example, simply the set $\mathbb{Z}$ of all integers, $\{3z+1 \mid z \in \mathbb{Z}\}$, or even $\{\pm 10^z \mid z \in \mathbb{Z}\}$ can be soundly adopted as the set of exact numbers.

From a practical point of view, on the other hand, choosing a set as the set of exact numbers needs a few more considerations since the arithmetic precision and conservative force for exactness of our number system depends on the set we choose; the arithmetic precision of our number system strictly depends on the *density* of the chosen set of exact numbers. The density of a countably infinite subset of $\mathbb{R}$ is the counterpart of the arithmetic density (asymptotic density) of a subset of $\mathbb{N}$. For example, the set $\mathbb{Q}$ of all rational numbers is *more dense* than $\mathbb{Z}$ even though $|\mathbb{Q}| = |\mathbb{Z}|$, since there exist infinitely many elements in $\mathbb{Q}$ which are not in $\mathbb{Z}$ while any element in $\mathbb{Z}$ is also an element of $\mathbb{Q}$.

The closure property of the set of exact numbers also matters due to the conservative force for exactness. If the chosen set E of exact numbers is closed under an n-ary arithmetic operation f on E^n, then the value of f is always in E regardless of its operands, conserving its exactness. On the contrary, if E is *not* closed under f, then the value of f may be *downgraded* into a proper interval or even an unwarranted number, according to its operands. For instance, suppose that $\mathbb{Z}$ is chosen as the set of exact numbers. In this setting, $5 \div 3$ cannot conserve its exactness any more even though 5 and 3 are exact numbers, since $\mathbb{Z}$ is not closed under division and $5 \div 3 \notin \mathbb{Z}$. In this case, the result has no choice but to be downgraded into an open interval $(1, 2)$, using integral (i.e., exact) ends. Even when multiplied by 6, this number cannot revive as an exact number and remains as an interval, $(6, 12)$. If the chosen set was $\mathbb{Q}$ at first, on the contrary, then this "downgrade" would not happen since $\frac{5}{1} \div \frac{3}{1} = \frac{5}{3}$ and $\frac{5}{3} \times \frac{6}{1} = \frac{10}{1}$.

In our previous work [16], the set $\mathbb{Q} = \{n/d \mid n \in \mathbb{Z},\ d \in \mathbb{N}^+,\ n \perp d\}$ of all rational numbers was adopted as the set of exact numbers, where $\perp$ denotes co-prime. In this paper, we extend this set by adding some portion of surd numbers, so-called "i-th root." Note that there exist not only irrational surd numbers but also rational surd numbers (e.g., $\sqrt{4}$), and that there are irrational numbers which are not surd (e.g., π). As a result, the set

$$E = \left\{ \frac{n}{d} \cdot \sqrt[i]{\frac{x}{y}} \middle| n \in \mathbb{Z},\ d, i, x, y \in \mathbb{N}^+, \right. \\ \left. (n \perp d) \wedge (x \perp y) \wedge (norm(i, x, y)) \right\}$$

is adopted as the set of exact numbers in this paper. The characteristic function of this set is actually the constructor of the canonical form for our exact numbers. The sign of n decides the sign of the corresponding exact number. The root-index i cannot be zero since it is a kind of denominator, i.e., $\sqrt[i]{r} = r^{\frac{1}{i}}$. It does not need to be negative since for every positive k and r, $\sqrt[-k]{r} = \sqrt[k]{\frac{1}{r}}$. We do not need to consider $r^{\frac{k}{i}}$ since it is the same with $\sqrt[i]{r^k}$. The radicand $\frac{x}{y}$ does not need to be zero, since if it is zero then the exact number itself is zero and should have the form of $(0/1) \cdot \sqrt[1]{1/1}$. We do not consider any negative radicand, since we only deal with real numbers. Finally, $norm(i, x, y)$ denotes whether $\sqrt[i]{x/y}$ is fully normalized, i.e., there is no rational number $q \neq 1$ and positive numbers j, a, b such that $\sqrt[i]{x/y} = q \cdot \sqrt[j]{a/b}$. In this paper, we sometimes denote an exact number e simply as a quintuple $\langle n, d, i, x, y \rangle$.

3.2 Proper intervals

We define the set I of all proper intervals based on the set E of exact numbers as follows:

$$I = \{\langle e_{\min}, e_{\max}, c_{\min}, c_{\max} \rangle \mid e_{\min}, \\ e_{\max} \in E,\ c_{\min}, c_{\max} \in \{\top, \bot\},\ e_{\min} < e_{\max}\}$$

where $e_{\min}$ and $e_{\max}$ denote the two ends of an interval, and $c_{\min}$ and $c_{\max}$ denote the closedness of those ends, respectively.

Here, the condition $e_{\min} < e_{\max}$ ensures that we do not take degenerate intervals into consideration, since degenerate intervals denote the same concept as exact numbers. This condition also implies that we do not take empty intervals into consideration, either. The condition $e_{\min}, e_{\max} \in E$ ensures that we consider neither unbounded intervals nor half-bounded intervals. Although these empty, unbounded, half-bounded intervals along with the concept of multiple intervals may also be useful in some specific examples such as calculating reciprocal of a proper interval containing zero, we have decided to exclude these aspects of intervals since they are beyond the main contribution of this work. From now on, whenever the term 'an interval' is used without any modifier, it means a proper interval.

3.3 Unwarranted numbers

Unlike exact numbers and intervals, unwarranted numbers do not need any fixed canonical form. The set U of unwarranted numbers is just a set of any numeric element which is not in E nor in I. "Any numeric element" here means any element of our set **RelNum** of all the reliable numbers. In other words, U is the relative complement of $E \cup I$ with respect to **RelNum**.

$$U = (E \cup I)^{\complement}$$

Owing to this flexible definition of U, **RelNum** can embrace any existing foreign numeric data types when implemented in an existing programming language such as Python.

4. ARITHMETIC OPERATIONS AND TYPE CONVERSIONS

In this section, we explain arithmetic operations on three classes defined in Section 3. In addition, we present how foreign datatypes are converted into our number system and we explain explicit type conversion of our number system.

4.1 Operations on exact numbers

The set E of exact numbers is closed under multiplication. It is also nearly closed under division, with one exception that division by zero is undefined.

For any two exact numbers $e_1 = \langle n_1, d_1, i_1, x_1, y_1 \rangle$ and $e_2 = \langle n_2, d_2, i_2, x_2, y_2 \rangle$, $e_1 + e_2 \in E$ if $i_1 = i_2 \wedge x_1 = x_2 \wedge y_1 = y_2$, since each of them is in its fully reduced form. For example, $2\sqrt{3} + 7\sqrt{3} = 9\sqrt{3}$. If at least one of those three components i, x, and y mismatches between two exact numbers e_1 and e_2,

then $e_1 + e_2$ should be downgraded to an unwarranted number, or to an interval if possible. The same goes for the subtraction.

For any exact number $e = \langle n, d, i, x, y \rangle$ and for any positive exact numbers $b \neq 1$, $b^e \in E$ if e is rational (i.e., $i = x = y = 1$). If e is not rational, b^e should be downgraded to an unwarranted number, or to an interval if possible.

It is very hard to find a set which is denser than $\mathbb{Z}$ and is closed under addition, subtraction, multiplication, division and exponentiation at once. We design E to be closed under multiplication rather than under addition and exponentiation, since multiplicative closure is easier to implement.

4.2 Operations on intervals and exact numbers

For any elementary arithmetic operation such as addition, subtraction, multiplication, division and exponentiation, if its operands are either exact numbers or intervals and at least one operand is an interval, then the operation is processed using the rules of the *basic interval arithmetic* [3]. These rules are basically operations between the ends of intervals, regarding inclusion of zero and negative numbers as some additional considerations. Hence, these are, in turn, the exact arithmetic described above since we used exact numbers for the ends of intervals.

The key character of our number system is the fact that "errors invade." In this point of view, exact numbers are more recessive than intervals, which are again more recessive than unwarranted numbers. If at least one operand is an interval and all the remaining operands are exact numbers, then the result falls back to an interval. For example, $3 + [2.4, 2.6)$ is not 5.5 but $[5.4, 5.6)$.

Note that this example does not mean an implicit coercion happens. In arithmetic operations where every operands are either exact numbers or intervals, implicit coercion never happens. This is a small difference between our number system and the basic interval arithmetic. In the basic interval arithmetic, the exact number 3 in the expression $3 + [2.4, 2.6)$ would be implicitly coerced into an interval $[3, 3]$ and then would be added to $[2.4, 2.6)$ using only 'interval addition.' In our number system, we cannot coerce 3 into $[3, 3]$ since we exclude degenerate interval. Instead, we define every combination of an interval and an exact operand for every binary operations separately.

4.3 Implicit coercions for foreign datatypes

The implicit coercion happens only when at least one operand has a type other than exact number and interval. For any operand which is neither an exact number nor an interval, our number system first tries to convert it into an exact number. In some programming languages, there exist some data types which can be regarded as exact numbers. For example, the unbounded integers `long` and `int` in Python can be safely converted into exact number without loss of their mathematical exactness.

If the operand of concern fails to be converted into an exact number, then we check whether it can be converted into an interval. For instance, `Decimal` class in Python is an approximation with additional information about error-range. Any number of type `Decimal` can be safely converted into an interval.

Finally, if the operand cannot be converted into an exact number or an interval, then the operand is regarded as an unwarranted number. In this case, all the other operands should be downgraded to unwarranted numbers. For example, the sum of an interval $[5.4, 5.6)$ and an IEEE 754 floating-point number 0.8 is not an interval $[6.2, 6.4)$ but a floating-point number 6.3.

4.4 Explicit type conversions

Our number system basically does not supply any generalized equipment for explicit type conversions, since most of them only breaks the reliability of number systems. For example, our number system prohibits any floating-point number from being converted to an interval or an exact number. This is the main difference between our number system and the `Decimal` class in Python, where a floating-point number can disguise itself as an exact decimal number.

5. LOGICAL OPERATIONS

In this section, we explain logical operations on three classes (i.e., exact numbers, proper intervals, and unwarranted numbers) described in Section 3. However, since the result of logical operations on intervals and unwarranted numbers is not always be decided, we need a three-valued logic to resolve the uncertainty. Hence, we first propose a three-valued logic. Based on the three-valued logic, we present logical operations between exact numbers and then logical operations between the rest of them.

5.1 Three-valued logic

We propose a three-valued logic system, where we can explicitly declare that something is *uncertain*. The main purpose of the three-valued logic is to resolve the problem that equalities and order relations involving intervals or unwarranted numbers cannot always be decided certainly. The set **TTV** of the three truth values is defined as follows:

$$\mathbf{TTV} = \{\texttt{True}_3, \texttt{Uncertain}, \texttt{False}_3\}$$

From a set-theoretic point of view, the set **TTV** clearly has three distinct elements in it. However, from a philosophical point of view, this set is different from ordinary sets with three elements. Note that the value `Uncertain` means not that "This is neither $\texttt{True}_3$ nor $\texttt{False}_3$," but that "This is indeed one of either $\texttt{True}_3$ or $\texttt{False}_3$." In other words, the `Uncertain` is not a completely distinct third value, but a not-yet-decided possibility to become one of the other two values.

TTV is a supertype of **Bool** since there exists an injection from **Bool** to **TTV**:

$$\{\top \mapsto \texttt{True}_3,\ \bot \mapsto \texttt{False}_3\}$$

which we can use as a coercion operation. Whenever a function over **TTV** is applied to a Boolean argument, that Boolean argument is *automatically* coerced into **TTV** by the injection above.

An element of **TTV**, however, is never converted into an element of **Bool** implicitly in any case. Any application of a function over **Bool** to an element of **TTV** should be prohibited as a type-error. For example, the '`if`' clauses in almost all programming languages should not accept **TTV** values as their conditional arguments. The main purpose of this intentional restriction is to avoid mistakes of programmers, especially by confusing the meaning of `else` blocks of `if` statements; in this case, `else` implies not only the falsity but also the uncertainty of the given condition.

Instead, we define three predicates over **TTV**, namely *surely*, *vague*, and *never*. In order for **TTV** values to be used in conditional judgments such as `if` or `while` clauses, they should be converted explicitly into **Bool** according to their accurate meanings. *Surely*, *vague* and *never* maps only $\texttt{True}_3$, `Uncertain` and $\texttt{False}_3$ to $\top$, respectively.

$$\begin{aligned} surely &= \{\texttt{True}_3 \mapsto \top,\ \texttt{Uncertain} \mapsto \bot,\ \texttt{False}_3 \mapsto \bot\} \\ vague &= \{\texttt{True}_3 \mapsto \bot,\ \texttt{Uncertain} \mapsto \top,\ \texttt{False}_3 \mapsto \bot\} \\ never &= \{\texttt{True}_3 \mapsto \bot,\ \texttt{Uncertain} \mapsto \bot,\ \texttt{False}_3 \mapsto \top\} \end{aligned}$$

Three-valued negation $\neg_3 : \mathbf{TTV} \to \mathbf{TTV}$ is defined as follows:

$$\neg_3 = \{\texttt{True}_3 \mapsto \texttt{False}_3,\ \texttt{Uncertain} \mapsto \texttt{Uncertain},\ \texttt{False}_3 \mapsto \texttt{True}_3\}$$

Note that the negation of `Uncertain` is `Uncertain` (i.e., "not uncertain" here does not mean "certain.") Since `Uncertain` denotes the concept of being "true or false," its negation denotes just "false or true."

Three-valued conjunction $\wedge_3 : \mathbf{TTV} \times \mathbf{TTV} \to \mathbf{TTV}$ is defined as follows:

$$x \wedge_3 y \text{ is } \begin{cases} \texttt{True}_3 & \text{if both } x \text{ and } y \text{ are } \texttt{True}_3 \\ \texttt{False}_3 & \text{if at least one of } x \text{ and } y \text{ is } \texttt{False}_3 \\ \texttt{Uncertain} & \text{otherwise} \end{cases}$$

Other three-valued logical operators, namely disjunction $\vee_3$, exclusive or $\otimes_3$, and implication $\to_3$ are defined as follows, using the operators defined above:

$$\begin{aligned} x \vee_3 y &\stackrel{def}{=} \neg_3(\neg_3 x \wedge_3 \neg_3 y) \\ x \to_3 y &\stackrel{def}{=} \neg_3 x \vee_3 y \\ x \leftrightarrow_3 y &\stackrel{def}{=} (x \to_3 y) \wedge_3 (y \to_3 x) \\ x \not\leftrightarrow_3 y &\stackrel{def}{=} \neg_3(x \leftrightarrow_3 y) \\ x \otimes_3 y &\stackrel{def}{=} (x \vee_3 y) \wedge_3 (x \not\leftrightarrow_3 y) \end{aligned}$$

5.2 Three-valued comparisons between two exact numbers

For an overloaded comparison operator over reliable numbers, there are nine non-overloaded cases according to the three classes of its two operands. We represent non-overloaded operators by the subscripts of two characters, each of which denotes the class of their two operands. For example, for the overloaded equality $=_{RR}$: **RelNum** $\times$ **RelNum** $\to$ **TTV**, there are nine non-overloaded equalities are follows:

$$\begin{aligned} =_{ee} &: E \times E \to \mathbf{TTV} \\ =_{ei} &: E \times I \to \mathbf{TTV} \\ =_{eu} &: E \times U \to \mathbf{TTV} \\ =_{ie} &: I \times E \to \mathbf{TTV} \\ =_{ii} &: I \times I \to \mathbf{TTV} \\ =_{iu} &: I \times U \to \mathbf{TTV} \\ =_{ue} &: U \times E \to \mathbf{TTV} \\ =_{ui} &: U \times I \to \mathbf{TTV} \\ =_{uu} &: U \times U \to \mathbf{TTV} \end{aligned}$$

where e, i and u denotes that the operand on the corresponding side of the equality is an exact number, an interval, and an unwarranted number, respectively.

Before defining overloaded comparison operators over reliable numbers, we first define non-overloaded operators over exact numbers. Overloaded operators and other non-overload cases involving intervals and unwarranted numbers will be defined later on the following subsections, using the operators over exact numbers defined here.

Though the common codomain of these operators is **TTV**, the common range is actually $\{\texttt{True}_3, \texttt{False}_3\}$ since every pair of exact numbers can always be compared without any uncertainty.

In the first place, we define the equality $=_{ee}$ using only the customary mathematical equality $=$ between integers. For any two exact numbers $\frac{n_1}{d_1} \cdot \sqrt[i_1]{\frac{x_1}{y_1}}$ and $\frac{n_2}{d_2} \cdot \sqrt[i_2]{\frac{x_2}{y_2}}$,

$$\frac{n_1}{d_1} \cdot \sqrt[i_1]{\frac{x_1}{y_1}} =_{ee} \frac{n_2}{d_2} \cdot \sqrt[i_2]{\frac{x_2}{y_2}} \text{ is } \begin{cases} \texttt{True}_3 & \text{if } n_1 = n_2 \wedge d_1 = d_2 \\ & \quad \wedge\, i_1 = i_2 \wedge \\ & \quad x_1 = x_2 \wedge y_1 = y_2 \\ \texttt{False}_3 & \text{otherwise} \end{cases}$$

since every exact number is, by definition, in its canonical form already. For any two exact numbers e_1 and e_2,

$$e_1 \neq_{ee} e_2 \stackrel{def}{=} \neg_3(e_1 =_{ee} e_2)$$

Similarly, the operator $<_{ee}$ is defined using only the customary mathematical inequalities $<$ and $\leq$ between integers. Let e_1 and e_2 be exact numbers $\frac{n_1}{d_1} \cdot \sqrt[i]{\frac{x_1}{y_1}}$ and $\frac{n_2}{d_2} \cdot \sqrt[k]{\frac{x_2}{y_2}}$, respectively. Then,

$$e_1 <_{ee} e_2 \text{ is } \begin{cases} \texttt{True}_3 & \text{if } (n_1 < 0 \wedge 0 \leq n_2) \vee \\ & \quad (0 \leq n_1 \wedge 0 \leq n_2 \wedge |n_1|^{ik} d_2^{ik} x_1^k y_2^i \\ & \qquad < |n_2|^{ik} d_1^{ik} x_2^i y_1^k) \vee \\ & \quad (n_1 < 0 \wedge n_2 < 0 \wedge |n_2|^{ik} d_1^{ik} x_2^i y_1^k \\ & \qquad < |n_1|^{ik} d_2^{ik} x_1^k y_2^i) \\ \texttt{False}_3 & \text{otherwise} \end{cases}$$

The operators $\leq_{ee}$, $>_{ee}$ and $\geq_{ee}$ can be simply derived from $<_{ee}$ and $=_{ee}$. For any exact numbers e_1 and e_2,

$$\begin{aligned} e_1 \leq_{ee} e_2 &\stackrel{def}{=} e_1 <_{ee} e_2 \ \vee_3\ e_1 =_{ee} e_2 \\ e_1 >_{ee} e_2 &\stackrel{def}{=} e_2 <_{ee} e_1 \\ e_1 \geq_{ee} e_2 &\stackrel{def}{=} e_2 \leq_{ee} e_1 \end{aligned}$$

5.3 Equalities between reliable numbers

The *overloaded* equality $=_{RR}$ over **RelNum**, in a set-theoretic point of view, is a union of nine *non-overloaded* equalities:

$$\begin{aligned} =_{RR} \quad &: \quad \mathbf{RelNum} \times \mathbf{RelNum} \to \mathbf{TTV} \\ =_{RR} \quad &\stackrel{def}{=} \quad =_{ee} \cup =_{ei} \cup =_{eu} \cup =_{ie} \cup =_{ii} \cup =_{iu} \cup =_{ue} \\ &\qquad \cup =_{ui} \cup =_{uu} \end{aligned}$$

since the set **RelNum** is partitioned into $\{E, I, U\}$. Note that the binary function $=_{RR}$ is neither a relation nor a binary predicate, since its codomain is not **Bool** but **TTV**. Among these nine non-overloaded equalities, $=_{ee}$ was already defined in the Section 5.2. We define remaining eight functions in this section.

Note that, though an interval is manipulated as if it is a line segment on the number line, it does not actually means a set of uncountably many numbers, but means a single number located somewhere inside that line segment. As a consequence, equalities involving intervals are matters of possibility and inevitability.

If an exact number resides within the boundary of an interval, we cannot decide whether they are equal or not. They are surely different, otherwise. Given an interval i, we denote its minimal end and maximal end as $\underline{e}(i)$ and $\overline{e}(i)$, respectively. Also, the closedness of its minimal end and maximal end are denoted as $\underline{C}(i)$ and $\overline{C}(i)$, respectively. Let $s : \mathbf{TTV} \to \mathbf{Bool}$ be an alias of the predicate *surely* defined earlier. Then,

$$x =_{ei} y \text{ is } \begin{cases} \mathtt{Uncertain} & \text{if } (s(\underline{e}(y) <_{ee} x) \vee \\ & \quad (s(\underline{e}(y) =_{ee} x) \wedge \underline{C}(y))) \wedge \\ & \quad (s(x <_{ee} \overline{e}(y)) \vee \\ & \quad (s(x =_{ee} \overline{e}(y)) \wedge \overline{C}(y))) \\ \mathtt{False}_3 & \text{otherwise} \end{cases}$$

and $=_{ie}$ is directly derived from it:

$$x =_{ie} y \stackrel{def}{=} y =_{ei} x$$

Two intervals are inevitably equal if and only if they refer to the same object. Two intervals cannot be equal if they do not share even one point on the number line. If two distinct interval objects share at least one point, then their equality is uncertain. Note that, if two intervals are not the same object, then we cannot say they are surely equal even when their ends and closedness coincide pairwise. In other words, though two intervals $[a, b]$ and $[a, b]$ have the same boundary notations, they may still be different as two approximated numbers.

$$x =_{ii} y \text{ is}$$

$$\begin{cases} \mathtt{True}_3 & \text{if both } x \text{ and } y \text{ refer to the same object} \\ \mathtt{False}_3 & \text{if } s(\overline{e}(x) <_{ee} \underline{e}(y)) \vee \\ & \quad (s(\overline{e}(x) =_{ee} \underline{e}(y)) \wedge \neg(\overline{C}(x) \wedge \underline{C}(y))) \vee \\ & \quad (s(\overline{e}(y) =_{ee} \underline{e}(x)) \wedge \neg(\overline{C}(y) \wedge \underline{C}(x))) \vee \\ & \quad s(\overline{e}(y) <_{ee} \underline{e}(x)) \\ \mathtt{Uncertain} & \text{otherwise} \end{cases}$$

Equalities involving at least one unwarranted number are always uncertain, since unwarranted numbers do not guarantee their own boundary at all.

$$\begin{aligned} x =_{ue} y \quad &\text{is} \quad \mathtt{Uncertain} \\ x =_{ui} y \quad &\text{is} \quad \mathtt{Uncertain} \\ x =_{ua} y \quad &\text{is} \quad \mathtt{Uncertain} \\ x =_{eu} y \quad &\stackrel{def}{=} \quad y =_{ue} x \\ x =_{iu} y \quad &\stackrel{def}{=} \quad y =_{ui} x \end{aligned}$$

The operator $\neq_{RR}$ is defined as the three-valued negation of $=_{RR}$:

$$x \neq_{RR} y \stackrel{def}{=} \neg_3(x =_{RR} y)$$

5.4 Orders between reliable numbers

We define four *overloaded* order relations, $<_{RR}$, $\leq_{RR}$, $>_{RR}$ and $\geq_{RR}$. Among these, $>_{RR}$ and $\geq_{RR}$ can be simply derived from the definitions of $<_{RR}$ and $\leq_{RR}$ respectively, since the latter are the converses of the former.

$$\begin{aligned} a <_{RR} b \quad &\stackrel{def}{=} \quad b >_{RR} a \\ a \leq_{RR} b \quad &\stackrel{def}{=} \quad b \geq_{RR} a \end{aligned}$$

Among the eighteen non-overloaded cases for the overloaded $<_{RR}$ and $\leq_{RR}$, we have already defined $<_{ee}$ and $\leq_{ee}$ in the Section 5.2. We define remaining sixteen cases in this section. The most notable characteristic of the orders involving intervals is that $\leq_{ei}$ cannot be derived from $<_{ei}$, $\vee_3$, and $=_{ei}$. For example, while $[a, b] \leq_{ei} b$ is surely true, $([a, b] <_{ei} b) \vee_3 ([a, b] =_{ei} b)$ is uncertain since both $[a, b] <_{ei} b$ and $[a, b] =_{ei} b$ are uncertain. $\leq_{ie}$ and $\leq_{ii}$ cannot be simply derived from $<_{ie}$ and $<_{ii}$ respectively, neither.

For any two intervals x and y, and for any exact number a,

$$a <_{ei} y \text{ is}$$

$$\begin{cases} \mathtt{True}_3 & \text{if } s(a <_{ee} \underline{e}(y)) \vee (s(a =_{ee} \underline{e}(y)) \wedge \neg\underline{C}(y)) \\ \mathtt{False}_3 & \text{if } s(\overline{e}(y) \leq_{ee} a) \\ \mathtt{Uncertain} & \text{otherwise} \end{cases}$$

$$a \leq_{ei} y \text{ is}$$

$$\begin{cases} \mathtt{True}_3 & \text{if } s(a \leq_{ee} \underline{e}(y)) \\ \mathtt{False}_3 & \text{if } s(\overline{e}(y) <_{ee} a) \vee (s(\overline{e}(y) =_{ee} a) \wedge \neg\overline{C}(y)) \\ \mathtt{Uncertain} & \text{otherwise} \end{cases}$$

$$x <_{ie} a \text{ is}$$

$$\begin{cases} \mathtt{True}_3 & \text{if } s(\overline{e}(x) <_{ee} a) \vee (s(\overline{e}(x) =_{ee} a) \wedge \neg\overline{C}(x)) \\ \mathtt{False}_3 & \text{if } s(a \leq_{ee} \underline{e}(x)) \\ \mathtt{Uncertain} & \text{otherwise} \end{cases}$$

$$x \leq_{ie} a \text{ is}$$

$$\begin{cases} \mathtt{True}_3 & \text{if } s(\overline{e}(x) \leq_{ee} a) \\ \mathtt{False}_3 & \text{if } s(a <_{ee} \underline{e}(x)) \vee (s(a =_{ee} \underline{e}(x)) \wedge \neg\underline{C}(x)) \\ \mathtt{Uncertain} & \text{otherwise} \end{cases}$$

$$x <_{ii} y \text{ is}$$

$$\begin{cases} \mathtt{True}_3 & \text{if } s(\overline{e}(x) <_{ee} \underline{e}(y)) \\ & \quad \vee (s(\overline{e}(x) =_{ee} \underline{e}(y)) \wedge \neg(\overline{C}(x) \wedge \underline{C}(y))) \\ \mathtt{False}_3 & \text{if } s(\overline{e}(y) \leq_{ee} \underline{e}(x)) \\ \mathtt{Uncertain} & \text{otherwise} \end{cases}$$

$$x \le_{ii} y \text{ is}$$

$$\begin{cases} \texttt{True}_3 & \text{if } s(\overline{e}(x) \le_{ee} \underline{e}(y)) \\ \texttt{False}_3 & \text{if } s(\overline{e}(y) <_{ee} \underline{e}(x)) \\ & \quad \vee \, (s(\overline{e}(y) =_{ee} \underline{e}(x)) \wedge \neg(\underline{C}(x) \wedge \overline{C}(y))) \\ \texttt{Uncertain} & \text{otherwise} \end{cases}$$

The remaining ten cases where at least one unwarranted number takes part in always return `Uncertain`.

$x <_{eu} y$ is `Uncertain`
$x <_{iu} y$ is `Uncertain`
$x <_{ue} y$ is `Uncertain`
$x <_{ui} y$ is `Uncertain`
$x <_{uu} y$ is `Uncertain`
$x \le_{eu} y$ is `Uncertain`
$x \le_{iu} y$ is `Uncertain`
$x \le_{ue} y$ is `Uncertain`
$x \le_{ui} y$ is `Uncertain`
$x \le_{ua} y$ is `Uncertain`

6. PROTOTYPE IMPLEMENTATION IN PYTHON

We implement the number classes and the three-valued logic described in Section 3, 4 and 5, as a module in Python. With this module, we can easily make operations on numbers and judgments on the conditions involving numbers more reliable. We also demonstrate a few cases where this module successfully prevents erroneous results.

6.1 Exact numbers

The class `exact` implements the set E of exact numbers. Five data attributes `exact.n`, `exact.d`, `exact.i`, `exact.x` and `exact.y` of the class `exact` correspond pairwise to the five components n, d, i, x and y of the canonical form $\frac{n}{d}\sqrt[i]{\frac{x}{y}}$ of an exact number. Recall that we assume we already have a set $\mathbb{Z}$ of unbounded integers along with its subset $\mathbb{N}^+$, when we defined the set E of exact numbers in Section 3.1. Since the components n, d, i, x and y above are unbounded (positive) integers, the corresponding data attributes should also be stored as unbounded integers internally. In Python, the built-in type `long` provides unbounded integers. Another built-in type `int`, though being the type of bounded integers basically, can also be regarded as a type of unbounded integers since `int` is automatically coerced into `long` whenever an overflow occurs. Hence, every data attribute of `exact` has type `int` or `long` in our implementation. From now on, we will call any value of type `int` or `long` an *integral* value, collectively.

The constructor of the class `exact` expects at least zero and at most five arguments:

```
class exact(object):
    def __init__(self, n=0, d=1,
                 i=1, x=1, y=1):
```

The last $0 \le k \le 5$ arguments omitted are replaced with their respective default values. For example, `exact(2,3)` is equivalent to `exact(2,3,1,1,1)`. Note that if no argument is given at all, i.e., `exact()`, it constructs an exact number zero, $\frac{0}{1}\sqrt[1]{\frac{1}{1}}$: this complies with the convention of Python saying "nullary call of a constructor should construct the zero-like element in that type."

The arguments supplied to the constructor are *normalized* before stored into their corresponding data attributes. This normalization ensures the condition which the set comprehension of E claims. For example, `exact(4,6)` constructs the same value with `exact(2,3)` since $\frac{4}{6} = \frac{2}{3}$. For another example, `exact(5,1,2,18)` constructs the same value with `exact(15,1,2,2)` since $5\sqrt{18} = 15\sqrt{2}$.

The constructor can also be applied to non-integral arguments, especially well-formed strings and other exact numbers: first, every well-formed string is, if any, parsed and translated into a corresponding exact number. Then, every exact number is integralized automatically.

If any non-integral argument is supplied to the constructor of `exact` (i.e., if there exists any argument of type `exact` or `str`), then the constructor automatically convert it into an integral value, through a process as follows. First, every argument of type `str`, if any, is checked whether it is syntactically well-formed or not. If not, the construction stops and it raises `ValueError`. Then, every well-formed string, if any, is parsed and translated into the corresponding exact number. Finally, every argument of type `exact`, if any, is resolved into a polynomial expression of integral values and rearranged with other arguments.

A well-formed string argument should have one of the following forms:

- $digit^\star(.digit^\star(_digit^+)^?)^?$
- $<digit^\star>digit^\star(.digit^\star(_digit^+)^?)^?$
- $digit^\star(.digit^\star(_digit^+)^?)^?<digit^\star>digit^\star(.digit^\star(_digit^+)^?)^?$

where

- *digit* is `[0-9]`,
- the underscore '_' means the start-point of the recurring part for a recurring decimal, and
- *<digit⋆>* means the base of root.

For example, the string `"1.33_428571"` denotes $1.33\dot{4}2857\dot{1}$ which is an exact (recurring) decimal representation of an exact number $\frac{467}{350}\sqrt[1]{\frac{1}{1}}$. The string `"<3>1._6"` denotes $\sqrt[3]{1.\dot{6}}$, an exact decimal representation of an exact number $\frac{1}{1}\sqrt[3]{\frac{5}{3}}$. Likewise, `"5<2>3"` denotes $5\sqrt{3}$, an exact number $\frac{5}{1}\sqrt[2]{\frac{3}{1}}$.

Once there remain only integers or exact numbers in the arguments, arguments of type `exact` are processed. For example,

$$\texttt{exact(exact(}n_n\texttt{, } d_n\texttt{, } i_n\texttt{, } x_n\texttt{, } y_n\texttt{), } d\texttt{, } i\texttt{, } x\texttt{, } y\texttt{)}$$

is rearranged into

$$\texttt{exact(}n_n\texttt{, } d_n d\texttt{, } i_n i\texttt{, } x_n^i x^{i_n}\texttt{, } y_n^i y^{i_n}\texttt{)}$$

since

$$\frac{\frac{n_n}{d_n}\sqrt[i_n]{\frac{x_n}{y_n}}}{d}\sqrt[i]{\frac{x}{y}} = \frac{n_n}{d_n d}\sqrt[i_n i]{\frac{x_n^i x^{i_n}}{y_n^i y^{i_n}}}$$

There is one restriction in this process for the third argument, i.e., the index of the root. Since we do not regard a number to the power of an irrational number, e.g., $3^{\sqrt{2}}$ as an exact number, the third argument should be rational. If it is irrational, then the construction stops and it raises `ValueError`.

6.2 Intervals and unwarranted numbers

The class `interval` implements the set I of proper intervals. Every instance i of the class `interval` has four data attributes i`.minend`, i`.maxend`, i`.mincls`, and i`.maxcls`, which correspond to $e_{\min}$, $e_{\max}$, $c_{\min}$, and $c_{\max}$ described in Section 3.2, respectively. Two flags of closedness i`.mincls` and i`.maxcls` should have type `bool`. Two ends i`.minend` and i`.maxend` should be instances of the class `exact`.

The constructor of the class `interval` expects at least two and at most four arguments, which correspond to the four data attributes described above.

```
class interval(object):
    def __init__(self, minend, maxend, \
                 mincls=True, maxcls=True):
```

The first two arguments are mandatory, and should be exact numbers or other types which can be automatically converted into exact numbers such as `int`, `long`, and `string`. The maximal end must be greater than the minimal end. The remaining two arguments should have the type `bool`. These optional arguments default to `True` if omitted, regarding that the corresponding end of the interval is closed.

We do not need any class for unwarranted numbers since in this version all the number types in Python except `int` and `long` are automatically regarded as unwarranted numbers in our implementation.

6.3 Three-valued logic

The class `ttv` implements the set **TTV** of three truth values defined in Section 5.1. Every instance t of the class `ttv` has only one data attributes t`.val`, which is one of -1, 0, or 1. Three constants are also defined as three instances of this class as follows:

```
True3     = ttv( 1)
False3    = ttv( 0)
Uncertain = ttv(-1)
```

We never define the reserved special method `__nonzero__()` in the `ttv` class, and hence *implicit* conversions from `ttv` to `bool` are strictly prohibited, in order to implement the restriction described in Section 5.1. Instead, three predicates on **TTV** are also implemented as follows:

```
def surely(t):
    return t.val == 1
```

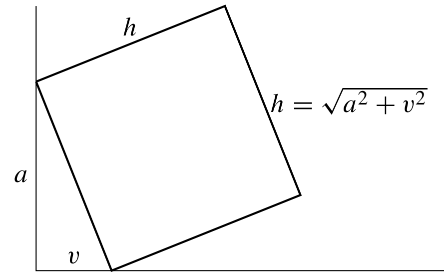

Figure 1 Area of the square whose edge is a hypotenuse of a right triangle.

```
def never(t):
    return t.val == 0

def vague(t):
    return t.val == -1
```

6.4 Case study

In Python, built-in floating-point numbers should be handled with great care since they do not behave as in the elementary mathematics. For example, the summation of ten *floating-point* 0.1's is not exactly 1. The program below cannot escape from the `while` loop, since the `count` does not exactly hit 2 but $1.999\cdots$.

```
count, offset = 1, 0.1
while True:
    count += offset
    if count == 2: break
```

The `while` loop above can be rewritten using our module as follows. The new program can escape from the loop since the summation of ten *exact* 0.1's is exactly 1, and the value of `count` is exactly after ten iterations.

```
from relnum import *
count, offset = 1, exact("0.1")
while True:
    count += offset
    if surely(count == 2): break
```

The exactness of surd numbers are important especially in many geometry problems. We also demonstrate a simplified example of this issue in Fig. 1; suppose that we already know a certain fixed length a, and that we repeatedly receive the measurement of variable length v from a sensor. In the example, the goal is to calculate the area of a square whose edge is the hypotenuse h of the right triangle whose remaining two edges are a and v.

In a mathematical point of view, the perfect answer for the area is simply $a^2 + v^2$ since it is equal to h^2. In computer programs, however, lots of calculations are modularized and each of those modules cannot guarantee the exactness of some operations such as exponentiation and n-th root. The following program cannot escape the loop at $v = 5.0 \pm 0.000001$ due to the error of square root involving built-in floating point numbers.

```
def hypotenuse(x, y):
    return ((x ** 2) + (y ** 2)) ** 0.5
def area_square(x):
    return x ** 2

while True:
    v = read_sensor_foo()
# value from CPS sensor
    if area_square(hypotenuse(4, v))
> 25.0: break
```

The following program, however, succeeds to escape at the same v, since the error inherent in the interval is controlled precisely during the operations. The hypotenuse and area are calculated as and compared with intervals rather than floating-point numbers, and the comparison ensures that the judgment does not leave any uncertainty:

```
from relnum import *

def hypotenuse(x, y):
    return ((x ** 2) + (y ** 2))
** exact(1, 2)
def area_square(x):
    return x ** 2

while True:
    v = read_sensor_foo()
# value from CPS sensor
    if surely(area_square(hypotenuse(4, v))
> \
         interval(25 - exact(0.001),
25 + exact(0.001))): break
```

7. CONCLUSION

We have designed and implemented a reliable number system to distinguish and separate any inexactness from the exactness. To guarantee certainties excluding any uncertainties resulted from the inexact numbers, we have also developed a corresponding three-valued logic. Our prototype implementation presents that we can avoid serious program errors, especially at the conditional branches where incorrect judgment of the equalities or orders of numeric values can occur.

For the future work, we want to develop this system further to include multiple intervals so that we can deal with reciprocals of intervals which contain zero. Moreover, the next implementation would be ported to Haskell [17] in order to type-check inappropriate numeric operations at compile time, and would be formalized in Coq [18] with dependent types.

Acknowledgements

This research was supported by the MSIT (Ministry of Science and ICT), Korea, under the ITRC (Information Technology Research Center) support program (2015-0-00445) supervised by the IITP (Institute for Information & communications Technology Promotion).

REFERENCES

1. P. Gowland and D. R. Lester, “A survey of exact arithmetic implementations,” in *Int’l Workshop on Computability and Complexity in Analysis*, pp. 30–47, 2000.
2. D. H. Bailey, “High-precision floating-point arithmetic in scientific computation,” *Computing in Science and Engineering*, vol. 7, pp. 54–61, 2005.
3. T. J. Hickey, Q. Ju, and M. H. van Emden, “Interval arithmetic: From principles to implementation,” *Journal of the ACM*, vol. 48, no. 5, pp. 1038–1068, 2001.
4. D. Hough, “Applications of the proposed IEEE-754 standard for floating point arithmetic,” *Computer*, vol. 14, no. 3, pp. 70–74, 1981.
5. Python Software Foundation, “Python v2.7.3 documentation.” `http://docs.python.org/`, 2012.
6. H. Putnam, “Three-valued logic,” *Philosophical Studies*, vol. 8, pp. 73–80, 1957.
7. C. Yap and T. Dube, “The exact computation paradigm,” *Computing in Euclidean Geometry, Lecture Notes Series on Computing*, vol. 4, pp. 452–492, 1995.
8. S. Wolfram, *The MATHEMATICA ® Book, Version 4*. Cambridge University Press, 1999.
9. MathWorks, “MATLAB The language of technical computing.” `http://www.mathworks.com/help/matlab/`, 2012.
10. S. van der Walt, S. Colbert, and G. Varoquaux, “The numpy array: A structure for efficient numerical computation,” *Computing in Science Engineering*, vol. 13, pp. 22 –30, march-april 2011.
11. T. Granlund, “GNU MP: The GNU Multiple Precision Arithmetic Library, Edition 5.0.5.” `http://gmplib.org/gmp-man-5.0.5.pdf`, 2012.
12. L. Fousse, G. Hanrot, V. Lefèvre, P. Pélissier, and P. Zimmermann, “MPFR: A multiple-precision binary floating-point library with correct rounding,” *ACM Trans. Math. Softw.*, vol. 33, June 2007.
13. R. R. Rajkumar, I. Lee, L. Sha, and J. Stankovic, “Cyber-physical systems: the next computing revolution,” in *Proceedings of the 47th Design Automation Conference*, DAC ’10, (New York, NY, USA), pp. 731–736, ACM, 2010.
14. W. Yan, S. Shi, Z. Liu, and G. Li, “Investigation on remote monitoring system for hybrid electric bulldozer,” *International Journal of Computer Systems Science and Engineering*, vol. 30, no. 5, 2015.
15. B. Jin, Q. Bai, H. Zhang, D. Wang, and Y. Gao, “Design and analysis of communication scheduling applied in water inrush perception layer of mine internet of things,” *International Journal of Computer Systems Science and Engineering*, vol. 30, no. 5, 2015.
16. R. Cha, W. Nam, and J.-Y. Choi, “Reliable integration of exact and approximated arithmetic with three-valued logic in python,” in *Proceedings of International Conference of Software Technology (SoftTech 2012)*, pp. 104–109, 2012.
17. S. Thompson, *Haskell: the craft of functional programming*. International computer science series, Addison Wesley, 1999.
18. Y. Bertot and P. Castéran, *Interactive Theorem Proving and Program Development: Coq’Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. An EATCS Series, Springer, 2004.