Reliable Integration of Exact and Approximated Arithmetic with Three-Valued Logic in Python*

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

[1] Korea university, Seoul 136-701 Korea
{reeseo,choi}@formal.korea.ac.kr
[2] Konkuk university, Seoul 143-701 Korea
wnam@konkuk.ac.kr

Abstract. The error-ranges of exact rational numbers and intervals can be guaranteed even during the arithmetic operations whereas we cannot rely on the error-ranges of floating-point numbers. In this paper, we propose a novel number system, where the exact rational numbers are strictly separated from inexact floating point numbers and carefully integrated with the inexact numbers. A three-valued logic is also shipped with our number system to appropriately deal with uncertainties due to the inexactness. A prototype implementation of our number system in Python is demonstrated.

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* is a dedicated approximation system where error-ranges are strictly guaranteed during the arithmetic operations [3]. These systems, however, are not so well integrated with unreliable approximations such as IEEE 754 floating-point numbers. For example, the class method `from_float` of the `Fraction` class that is a rational number type in Python [4], 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 into a IEEE 754 format [5]. Construction rules like this brake the reliability of the entire rational numbers in the fraction module of Python.

To confidently rely on the number systems, we claim that inexact numbers should be strictly distinguished from the exact numbers and intervals, and that the inexactness should invade 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.

* This research partially was supported by the MKE (Ministry of Knowledge Economy), Korea, under the ITRC (Information Technology Research Center) support program supervised by the NIPA (National IT Industry Promotion Agency): NIPA–2012–H0301–12–3006.

In this paper, hence, 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] where we can appropriately deal with uncertainties. First, we define three classes of numbers. Based on these classes, we define a set of coercion rules among them for the arithmetic operations and then define the equalities and order relations between them using three-valued logic. Finally, we demonstrate a prototype implementation in Python.

2 Constructions of Numeric Datatypes

In our number system, all the numbers are 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.
- 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. Each ends of an interval should be exact numbers, and can be either open or closed.
- an *inexact approximation* if its location or bounds cannot be guaranteed or cannot be represented exactly. A number in this class contains only blurred, unreliable information about its location.

2.1 Exact numbers

Exact numbers are constructed using `exact()` which expects at most two arguments and returns a *reduced* form of a rational number. The first argument is a numerator, and defaults to 0 of type `int` if omitted. The second one is a denominator, and defaults to 1 of type `int` if omitted. The arguments of `exact()`, if any, should have a type `int`, `long`, `string`, or another `exact`. If any of its argument is a string, it should have the form: $\mathit{digit}^\star(.\mathit{digit}^\star(\_\mathit{digit}^+)^?)^?$, where *digit* is `[0-9]` and the underscore '_' means the recurring decimals. For example, the string `"1.33_428571"` means $1.33\dot{4}28571\dot{}$ and `exact("1.33_428571")` constructs a rational number $\frac{467}{350}$, which can also be constructed by `exact(467,350)`.

2.2 Proper intervals

Proper intervals are constructed using `interval()` which expects at least two and at most four arguments. The first two mandatory arguments mean its minimal and maximal ends, 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 are the closedness of two ends, and should have the type `bool`. These

optional arguments default to `True` if omitted, regarding the interval is closed. We do not deal with degenerate intervals since they are equivalent to the corresponding exact numbers and are not approximations conceptually. Although we currently do not deal with unbounded, half-bounded, empty, or multiple intervals, these intervals can be included in the future developments in order to permit, for example, a reciprocal of $[-1, 1]$ which is $\{(-\infty, -1], \texttt{NaN}, [1, \infty)\}$.

2.3 Inexact approximations

Inexactly approximated numbers are constructed using `approx()`, which expect at most one argument defaulting to 0.0 of the type `float`. Every IEEE 754 compatible floating-point numbers fall back to this category. Some exact numbers which cannot be constructed consistently also fall back here. This class resembles the `decimal` module of Python [4], but is regarded as *unreliable*.

2.4 Coercions during arithmetic operations

When an arithmetic expression contains two or more different types of numbers, one of them is coerced to another. The basic rule of coercions within our number system reflects the fact that "errors invade." In this point of view, exact numbers are more recessive than intervals, which are again more recessive than approximations. If at least one operand in an arithmetic expression is an approximation, then the others fall back to an approximation. 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 $[5.4, 5.6)$. Similarly, $[5.4, 5.6) +$ `approx(0.8)` is not $[6.2, 6.4)$ but `approx(6.3)`. When numbers beyond our number system are mixed with at least one number in our number system, they are coerced into numbers of our number system as follows: `int` and `long` are coerced into exact numbers, and `Decimal` is coerced into an interval, and the others such as `float` are coerced into inexact approximations.

3 Logical Operations

3.1 Three-valued logic

For intervals and inexact approximations, their equalities or order relations cannot be guaranteed. Hence, we propose a three-valued logic system where we can explicitly declare that something is *uncertain*. The `TVL` class consists of three distinct values:

```
TVL := inevitable | uncertain | impossible
```

We *never* define the `__bool__()` method for the `TVL` class to avoid mistakes of programmers, especially by confusing the meaning of `else` block of `if` statements. Instead, to use these TVL values in the conditional judgments such as `if`

or `while` statements, we define three predicates namely `inevitably()`, `never()`, and `uncertain()`, of the type `TVL` $\rightarrow$ `bool`.

For the coercion in logical expressions, if a logical expression contains at least one number of our number system, then the other numbers are coerced in the same way described in the previous section.

3.2 Equalities

We implement *overloaded* operators $=$ and $\neq$ using two special methods `__eq__()` and `__ne__()`. There are six cases for the comparison of two numbers since equalities are symmetric:

- Two exact numbers are always inevitably equal or never equal without any uncertainty.
- An exact number e and an interval i cannot be inevitably equal since we rule out degenerate intervals. Their equality is uncertain if e is equal to one of the closed ends of i or e is in the range of i, and impossible otherwise.
- An exact number e and an approximation a are inevitably different if the exponents of e in the IEEE 754 form is different from that of a. In all the other cases, their equalities are uncertain.
- Two intervals are inevitably equal if they are the same instances of Python classes. If they have no intersection at all, then they are inevitably different. In all the other cases, their equalities are uncertain.
- An interval i and an approximated number a cannot be inevitably equal. If a is inevitably different from any of the two ends of i, and a is not in the range of i, then they are inevitably different. In all the other cases, their equalities are uncertain.
- The equality of two approximations is uncertain if their significant digits and exponents are equal; otherwise, they are inevitably different.

3.3 Order relations

For the *overloaded* order relation $<$, there are nine cases according to the three classes of its two operands, since this relation is not symmetric. Furthermore, especially for the intervals, $a \leq b$ does not always coincide with $a < b \vee_{\mathtt{T}} a = b$ where $\vee_{\mathtt{T}}$ is the disjunction operator for our three-valued logic. So, we should define overloaded $\leq$ for those nine cases separately. Fortunately, overloaded $>$ and $\geq$ are still the converse relations of $<$ and $\leq$, respectively. We omit the definitions for the orders of two exact numbers $<_{ee}$ and $\leq_{ee}$, since they are obvious without any uncertainty. Using these two relations and their converses, along with the equality $=_{ee}$ defined in Section 3.2, we formalize the order of two intervals, $<_{ii}$ and $\leq_{ii}$.

Let $E_{\min}$ and $E_{\max}$ be functions from `interval` to `exact`, each of which maps an interval to its minimal end and maximal end, respectively. Let $C_{\min}$ and $C_{\max}$ be predicates from `interval` to `bool`, each of which maps an interval to the closedness of its minimal end and maximal end, respectively. Let I : `TVL` $\rightarrow$ `bool`

be the predicate `inevitably()` defined earlier. Then, for any two intervals a and b,

$$a <_{ii} b \mapsto \begin{cases} \texttt{inevitable} & \text{if } I(E_{\max}(a) <_{ee} E_{\min}(b)) \\ & \quad \vee (I(E_{\max}(a) =_{ee} E_{\min}(b)) \wedge \neg(C_{\max}(a) \wedge C_{\min}(b))) \\ \texttt{impossible} & \text{if } I(E_{\max}(b) \leq_{ee} E_{\min}(a)) \\ \texttt{uncertain} & \text{otherwise} \end{cases}$$

Similarly,

$$a \leq_{ii} b \mapsto \begin{cases} \texttt{inevitable} & \text{if } I(E_{\max}(a) \leq_{ee} E_{\min}(b)) \\ \texttt{impossible} & \text{if } I(E_{\max}(b) <_{ee} E_{\min}(a)) \\ & \quad \vee (I(E_{\max}(b) =_{ee} E_{\min}(a)) \wedge \neg(C_{\min}(a) \wedge C_{\max}(b))) \\ \texttt{uncertain} & \text{otherwise} \end{cases}$$

Orders between other combinations of classes such as $<_{ie}$ are also omitted in this paper, since these are more obvious than $<_{ii}$.

4 Prototype Implementation

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
```

We have implemented the number classes and the three-valued logic described in Section 2 and 3, as a module in Python. With this module, we can easily make numbers more reliable. 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 `count` exactly hits 2 after ten iterations.

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

5 Conclusion

We have designed and implemented a novel number system to distinguish and separate any inexactness from the exactness. We have also developed a corresponding three-valued logic, to guarantee certainties excluding any uncertainties

resulted from the inexact numbers. Our prototype implementation shows 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 will develop this system further to include algebraic surd numbers and exponential operations on them. The extension will also include multiple intervals so that we can deal with reciprocals of intervals which contain zero. Moreover, the next implementation will also be ported to Haskell in order to type-check inappropriate numeric operations at compile time, and will be formalized in Coq with dependent types.

References

1. Gowland, P., Lester, D.R.: A survey of exact arithmetic implementations. In: International Workshop on Computability and Complexity in Analysis. (2000) 30–47
2. Bailey, D.H.: High-precision floating-point arithmetic in scientific computation. Computing in Science and Engineering **7** (2005) 54–61
3. Hickey, T.J., Ju, Q., van Emden, M.H.: Interval arithmetic: From principles to implementation. Journal of the ACM **48**(5) (2001) 1038–1068
4. Python Software Foundation: Python v2.7.3 documentation. (2012) `http://docs.python.org/`.
5. Hough, D.: Applications of the proposed ieee-754 standard for floating point arithmetic. Computer **14**(3) (1981) 70–74
6. Putnam, H.: Three-valued logic. Philosophical Studies **8** (1957) 73–80