The Ruby library *long-decimal*

Karl Brodowsky
IT Sky Consulting GmbH
Switzerland

2011-02-06

Abstract

The goal of the Ruby-library *long-decimal* is to provide a new numerical type *LongDecimal* and arithmetic operations to deal with this type.

1 Introduction

Ruby supports non-integral numbers with the built-in types *Float* and *Rational* and *BigDecimal*. While these are very useful for many purposes, the development of finance application requires the availability of a type like LongDecimal, that allows explicit control of the rounding and uses a decimal representation for the fractional part. An instance of *LongDecimal* can be represented as a pair of integral numbers (n, d), where the value of the number is $\frac{n}{10^d}$. The ring-operations $+$, $-$ and $*$ can be trivially defined in a way that does not loose any information (and thus does not require any rounding).
Division requires some additional thought, because the exact quotient of two *LongDecimal*s can very well be expressed as Rational, but not always as *LongDecimal*. A supplementary numeric type *LongDecimalQuot* has been introduced to support storing pairs (r, d), where r is a rational number and d is an estimation about the significant digits after the decimal point. The represented value is the rational number r.
Calculation of roots and trancendental functions require additional information as input parameters to control the number of digits after the decimal point and the rounding rules for achieving the result.
In the remainder of this document we will write $\mathrm{l}(n, d)$ for the *LongDecimal*-number represented by the pair (n, d) in the manner described above and $\mathrm{q}(r, d)$ for the *LongDecimalQuot*-number represented by teh pair (r, n). It needs to be observed that the part d in both cases carries information about the precision of the number in terms of significant digits after the decimal point.

2 Ring Operations

The ring operations $+$, $-$ and $*$ and obvious derived operations like negation are defined like this:

$$\bigwedge_{m,n\in\mathbb{N}_0}\bigwedge_{d,e\in\mathbb{Z}} \mathrm{l}(m,d)+\mathrm{l}(n,e)=\mathrm{l}(m\frac{\max(d,e)}{d}+n\frac{\max(d,e)}{e},\max(d,e))$$

$$\bigwedge_{m,n\in\mathbb{N}_0}\bigwedge_{d,e\in\mathbb{Z}} \mathrm{l}(m,d)-\mathrm{l}(n,e)=\mathrm{l}(m\frac{\max(d,e)}{d}-n\frac{\max(d,e)}{e},\max(d,e))$$

$$\bigwedge_{m,n\in\mathbb{N}_0}\bigwedge_{d,e\in\mathbb{Z}} \mathrm{l}(m,d)\cdot\mathrm{l}(n,e)=\mathrm{l}(mn,d+e)$$

3 Rounding Operations

The library supports two rounding operations that can be applied both to *LongDecimal* and *LongDecimalQuot*. In the latter case it implies a conversion to *LongDecimal*. The normal rounding function *round_to_scale* changes the precision to the given value, while keeping the value expressed by the number approximately the same.
The optional second parameter describes how rounding will be done, if the process looses information. It defaults to *ROUND_UNNECESSARY*, in which case an exception is raised whenever a value changing rounding process would be required.

rounding mode	**description**
ROUND_UP	always round in such a way that the absolute value does not decrease.
ROUND_DOWN	always round in such a way that the absolute value does not increase.
ROUND_CEILING	always round in such a way that the rational value does not decrease.
ROUND_FLOOR	always round in such a way that the rational value does not increase.
ROUND_HALF_UP	round to the nearest available value, prefer increasing the absolute value if the last digit is 5
ROUND_HALF_DOWN	round to the nearest available value, prefer decreasing the absolute value if the last digit is 5
ROUND_HALF_CEILING	round to the nearest available value, prefer increasing the rational value if the last digit is 5
ROUND_HALF_FLOOR	round to the nearest available value, prefer decreasing the rational value if the last digit is 5
ROUND_HALF_EVEN	round to the nearest available value, prefer the resulting last digit to be even if the last digit prior to r
ROUND_UNNECESSARY	raise an exception if value changing rounding would be needed

In addition to these commonly available rounding operations *long-decimal* provides rounding to remainder sets. This is motivated by the practice in some currencies to prefer using certain multiples of the smallest unit. For example in CHF you commonly use two digits after the decimal point, but the last digit is required to be 0 or 5. This is achieved as a special case of a more general concept *round_to_allowed_remainders*. A modulus $M \geq 2$ and a set $R \subset \mathbb{N}_0$ are given and a number x is rounded to $\mathrm{l}(n, d)$ such that

$$\bigvee_{r \in R} n \equiv r(\mod M)$$

Typically M is ten or a power of ten, but this is not required. Neither is it required that zero is a member of R. In that case an additional parameter is needed in order to define to which direction a potential last digit of zero would be rounded. In the CHF case we would have $M = 10$ and $R = \{0, 5\}$.

4 Division

Regular division using / yield an instance of *LongDecimalQuot*. The approximate number of significant digits is estimated by using the partial derivatives of $f(x,y) = \frac{x}{y}$. Assume $x = \mathrm{l}(m,s)$ and $y = \mathrm{l}(n,t)$. So we get for $\mathrm{l}(m,s)/\mathrm{l}(n,t)$ an result

$$\frac{x}{y} = \mathrm{q}(10^{t-s}\frac{m}{n}, r)$$

with

$$r \approx -\log_{10}\left(10^{-s}\frac{1}{|y|} + 10^{-t}\frac{|x|}{y^2}\right) = 2\log_{10}(y) + s + t - \log_{10}\left(|m| + |n|\right))$$

In order to avoid expensive logarithmic operation for such basic operations as division this is approximated by

$$r = \max(0, 2v + s + t - \max(u + s, v + t) - 3)$$

with $u = \lfloor \log|m| \rfloor - s + 1$ and $v = \lfloor \log|n| \rfloor - t + 1$ for m and n not zero and $u = -s$ for $m = 0$ and $v = -t$ for $n = 0$. Using $\max(a,b)$ instead of $log_{10}(10^a + 10^b)$ is a good approximation, when a and b are far apart.
It is recommended to use explicit rounding after having performed division rather than to rely on this somewhat arbitrary estimation on the number of significant digits. It is possible to retain intermediate results with *LongDecimalQuot*, because the full arithmetic is available for this type as well, but using rational numbers internally it will blow up numerator and denominator to an extent that diminishes performance by quite a margin in longer calculations.

5 Roots

Square root and cube root of LongDecimal can be calculated quite efficiently using an algorithm that is somewhat similar to the algorithm for long integer division. This has been preferred over the more commonly known Newton algorithm. Since square and cube roots are usually irrational, it is mandatory to provide rounding information concerning the number of desired digits and the rounding mode.

6 π

The number π can be calculated to a given number of digits, which works sufficiently fast for a few thousend digits.
Please use dedicated programs if you seriously want to calculate π to millions of digits, Ruby is not fast enough for this kind of number crunching to compete with the best C-programs.

7 Transcendental Functions

It is the goal of this library to support the most common transcendental functions in the long run. Currently $\exp$ and $\log$ are supported. These are calculated using the Taylor series, but the calculation has been improved. Most important it is to improve the convergence, which is the case with a series of the form

$$\sum_{n=0}^{\infty} \frac{x^n}{n!}$$

for $x < 0.5$.
For the exponential function this can always be achieved by using the following transformations: $x = 2^n x_0$ with $x_0 < 0.5$ implies

$$\exp(x) = \exp(x_0)^{2^n}$$

which can be easily calculated using successive squaring operations.
For the logarithm we first use $x = e^n x_0$ with $n \in \mathbb{N}_0$ and $x_0 < e$ with

$$\log(x) = n + \log(x_0).$$

From here we make use of the efficient square root calculation facility and use

$$\bigwedge_{m=0}^{\infty} x_{m+1} = \sqrt{x_m}$$

and

$$\log(x) = n + 2^m \log x_m$$

Using this x_m can be brought sufficiently close to 1 to make the Taylor series of $\log$ converge sufficiently fast to be useful for the calculation:

$$\log(x_m) = \sum_{k=1}^{\infty} \frac{(x_m - 1)^k (-1)^{k+1}}{k}$$