A Specification of the Non-Integral Calculations in the Ledger

Matthias Güdemann
matthias.gudemann@iohk.io
May 22, 2019

Abstract
This document defines a way to exactly calculate non-integral calculations in the ledger for Shelley which use elementary mathematical functions. The main objective is to give an unambiguous specification that gives the same results, independent of the architecture or programming language. The goal is to prevent chain forks because of slight differences in calculated results.

Contents
1 Introduction 2
2 Problem Description 2
3 Implementation Possibilities 2
   3.1 IEEE 754 Floating Point ......................................................... 2
   3.2 Rational Numbers ................................................................. 2
   3.3 Fixed Point Arithmetic .......................................................... 3
   3.4 Conclusion ................................................................. 3
4 Approximation Algorithms 3
   4.1 Taylor Series ................................................................. 3
   4.2 Continued Fractions .............................................................. 4
   4.3 Scaling ................................................................. 4
   4.4 Convergence ................................................................. 5
   4.5 Conclusion ................................................................. 5
5 Reference Implementation 5
6 Optimisations for Specific Use-Cases 6
References 6

List of Figures
1 Relative Error for Taylor Series Approximation of ln ......................... 8
2 Relative Error for Continued Fraction Approximation of ln .............. 9
3 Relative Error for Taylor Series Approximation of exp .................. 10
4 Relative Error for Continued Fraction Approximation of exp .......... 11
5 Benchmark Results for the Haskell Implementation ....................... 12
1 Introduction

The Shelley specification Formal Methods Team (TODO) and the Ouroboros protocol Kiayias et al. (2016) use non-integral calculations. Most can be done in an exact way using rational numbers and arbitrary precision integers which are available in most programming languages or via external libraries.

There are also instances where elementary functions are used which cannot be represented as rational numbers and therefore require a different treatment. This includes using the exponential function $e^x, x \in \mathbb{R}$ to calculate a decay and non-integral exponentiation $x^y, x, y \in \mathbb{R}$.

2 Problem Description

The calculations that involve elementary functions are concerned with decay of the value to refund from a deposit, calculation of the pool reward via the moving average and the leader election probability.

In all these cases, it is important that all distributed nodes calculate the same values, else here might be a disagreement what to include in the blockchain and forks might result. In the case of a single implementation, this will not be a problem, as only one result is possible, but in the case of multiple implementations, this can become an issue. As Cardano is striving to be a specification defined cryptocurrency, we need an exact specification of how to achieve this, allowing for independent implementations with the same behavior.

3 Implementation Possibilities

There are three main different possibilities to implement non-integral calculations. In Haskell, we can use typeclasses to design generic algorithms which can handle different backends. Each back-end has its advantages and disadvantages.

3.1 IEEE 754 Floating Point

A straight-forward approach is to use IEEE 754 floating-point arithmetic which is generally supported by all relevant programming languages and CPU architectures.

The basic arithmetic operations of IEEE 754 floating-point are well specified, and they are very efficient due to direct HW implementation.

Double precision floating-point provides 53 bits (15.99 decimal digits) of precision which is slightly below the required precision to represent the fraction of 1 lovelace, as there are $4.5 \cdot 10^{16}$ of them. Also, the required elementary functions are not standardized and there can be differences in different implementations. It is also worth noting that some architectures provide excess precision $^2$, which can result in slight differences in results. There exist programming languages, e.g., D, which make use of this by default. Many languages support excess precision but default to IEEE 754 64 bits, in particular on amd64 most make use of SSE floating-point arithmetic. Another source problems can be different rounding modes which some programming languages allow changing, e.g., C, but others do not, e.g., Haskell.

3.2 Rational Numbers

Rational numbers can be implemented on top of exact arbitrary precision integral arithmetic. Effectively one uses $gcd$ calculation to normalize the results after each basic operation. This

---

$^1$The most widely used library is the GNU multi-precision library (GMP)

$^2$The original x87 provided 80 bit floating-point
allows for arbitrary precise approximation of all real numbers, but may also incur arbitrary long
numerators or denominators.

These arbitrarily long numerators and denominators incur non-predictable run-time.

### 3.3 Fixed Point Arithmetic

An alternative approach is to use fixed point arithmetic based on exact integers where a propor-
tion of the integer is used as fractional part. This prevents the problem with using fully rational
arithmetic which can get very inefficient and allows to define the desired number of decimal
digits.

As it is implemented purely in SW on top of exact integer arithmetic, on the one hand it is
less efficient than IEEE 754, on the other hand it can be made to behave equivalently in different
implementations.

The basic idea is the following: for $n$ decimal digits of precision, multiply each integral value
with $10^n$ and take this into account in the basic operations. To use in the case of Cardano, this
would mean using $n = 17$ to support the required 17 digits of precision to track each possible
stake fraction.

### 3.4 Conclusion

For the Cardano cryptocurrency, the approach based on fixed point arithmetic seems to be the
best adapted one. It allows for the desired precision and can be implemented in an equivalent
way, because it uses exact integer arithmetic. While the fixed-point approach is less efficient
than IEEE 754, using partial pre-computation will be possible to speed it up in the case where
the base of an exponentiation is a constant.

### 4 Approximation Algorithms

The elementary functions required in the ledger are the exponential function and the expo-
nentiation. Exponentiation $x^y$ of arbitrary real numbers is calculated via the identity: $x^y = \exp(\ln(x^y)) = \exp(y \cdot \ln(x))$. This means we do need approximation schemes for $\exp(x)$ and $\ln(x)$.

There exist two main approaches for approximation: Taylor / MacLaurin series and contin-
ued fractions. Both require only basic arithmetic operations and allow for iterative approxima-
tion, i.e., constructing a sequence of $x_0, x_1, \ldots x_n$ where each $x_i$ is a better approximation for the
desired value.

#### 4.1 Taylor Series

A Taylor series defines an infinite series that approximates an infinitely often differentiable
function around a point $a$. It does have the following general form for a function $f$:

$$T_f(x; a) := \sum_{n=0}^{\infty} \frac{f^{(n)}(a)}{n!} (x - a)^n$$

Most often one develops the Taylor series at $a = 0$ which is also called MacLaurin series and
uses a truncated, finite polynomial for the function approximation as follows:

$$x_m := \sum_{n=0}^{m} \frac{f^{(n)}(0)}{n!} x^n$$

---

3 potentially 1s for basic arithmetic operations
4 There already exist two implementations, Haskell and C.
Using the above, one approximates the exponential function using Taylor series as

\[ \exp(x) := \sum_{n=0}^{\infty} \frac{x^n}{n!} \]

and the natural logarithm as

\[ \ln(x) := \sum_{n=1}^{\infty} \frac{(-1)^{n+1}}{n} (x - 1)^n \]

### 4.2 Continued Fractions

Continued fractions are a way to represent a number as the sum of its integral part and the reciprocal of another number. The most general form looks like this:

\[
\frac{b_0}{b_1 + \cfrac{a_1}{b_2 + \cfrac{a_2}{b_3 + \ddots}}}
\]

The convergents \( x_i = \frac{A_i}{B_i} \) are computed via the following recursion:

\[
A_{-1} := 1 \\
A_0 := b_0 \\
B_{-1} := 0 \\
B_0 := 1 \\
A_n := b_n \cdot A_{n-1} + a_n \cdot A_{n-2} \\
B_n := b_n \cdot B_{n-1} + a_n \cdot B_{n-2}
\]

For the exponential function \( \exp(x) \), the sequences of \( a_i \) and \( b_i \) are the following:

\[
\sigma(a_i) := 1 \cdot x, -1 \cdot x, -2 \cdot x, \ldots \\
\sigma(b_i) := 1, 2, x, 3, x, 4, x, 5, x, \ldots
\]

For the natural logarithm \( \ln(x + 1) \), the sequences of \( a_i \) and \( b_i \) are the following:

\[
\sigma(a_i) := 1^2 \cdot x, 1^2 \cdot x, 2^2 \cdot x, 2^2 \cdot x, 3^2 \cdot x, 3^2 \cdot x, \ldots \\
\sigma(b_i) := 1, 2, 3, \ldots
\]

### 4.3 Scaling

Both Taylor series and continued fractions do not converge for arbitrary input values, their convergence radius is limited. Therefore we apply scaling before we do the approximation, using the mathematical properties of \( \exp \) and \( \ln \) to get the correct results.

To calculate the exponential functions, we scale \( x \) in such a way that \( x \in [0; 1] \) via:

\[ \exp(x) = \exp \left( \frac{x}{n} \cdot n \right) = \left( \exp \left( \frac{x}{n} \right) \right)^n, \text{ with } n := \lceil x \rceil \]

For the natural logarithm, we calculate \( n \) in such a way that \( \exp(n) \leq x < \exp(n+1) \) and use this as follows:
\[ \ln(x) = \ln \left( \exp(n) \cdot \frac{x}{\exp(n)} \right) = \ln(\exp(n)) + \ln \left( \frac{x}{\exp(n)} \right) = \ln(\exp(n)) + n + \ln \left( \frac{x}{\exp(n)} \right) \]

which guarantees that \( \frac{x}{\exp(n)} \) is in the interval \( [1; e] \) which lies in the convergence radius of the approximation schemes.

### 4.4 Convergence

Experimental results have shown that continued fractions have a better convergence speed, in particular for the natural logarithm.

Figure 1 shows the relative approximation error for \( \ln \) with 10, 20, and 50 iterations using a Taylor series. Figure 2 shows the relative approximation error for 10, 20, and 50 iterations using continued fractions. In this case, the error of the continued fraction approach is multiple orders of magnitude lower for the same number of iterations.

Figure 3 shows the relative approximation error of 10 and 20 iteration using a Taylor series approach. Figure 4 shows the relative error of 10 and 20 iterations using continued fractions. In this case the error of continued fractions is around one order of magnitude lower for the same number of iterations.

### 4.5 Conclusion

From these experiments one can conclude that the convergence speed is higher for continued fractions, in particular for the natural logarithm. On the other hand, timing analysis showed that due to the simpler calculation per iteration of Taylor series, the exponential function can be approximated more efficiently using that approach. Therefore we decided to use both, Taylor series for \( \exp \) and continued fractions for \( \ln \).

To decide when the approximation has enough precision, we use the following criterion for two succeeding approximations \( x_n, x_{n+1} \):

\[ |x_n - x_{n+1}| < \epsilon \]

Therefore both approximation schemes result in the same precision.

### 5 Reference Implementation

The continued fraction approach using fixed point arithmetic has been implemented in Haskell and in C using the GNU multi-precision library \(^5\). For the Haskell version there are Quickcheck property-based tests that validate mathematical consistency of the laws for \( \ln \) and \( \exp \).

For both the C and the Haskell version there exists a test program that reads two fixed pointed numbers with 34 decimal digits of precision and calculates \( x^y \) from them. This has been used to validate the two different implementations for 20,000,000 randomly generated testcases where \( x \) and \( y \) were drawn uniformly from the interval \( [0.1; 100.1] \), which showed that all results were exactly the same for all 34 decimal digits.

The important aspect to get equivalent results is using the same approach to rounding after a division. In particular we use rounding to \( -\infty \).

---

\(^5\) https://gmplib.org
6 Optimisations for Specific Use-Cases

One use case for non-integral calculations in Shelley is the calculation of the probability of being a slot leader. This calculation has to be done every 2s locally for each potential slot leader. It also needs to be done in order to validate a block (to make sure that the block producer actually had the right to do so).

More precisely, one has to check whether for a given \( p \), \( \sigma \) and a constant (at least for one epoch) \( f \) the following inequality holds:

\[
p < 1 - (1 - f)^\sigma
\]

As \( 1 - f \) is considered to be constant, and because \( (1 - f)^\sigma \) is equal to \( \exp(\sigma \cdot \ln(1 - f)) \), we can pre-compute the value of \( \ln(1 - f) \) once and use it for every following computation.

Setting \( c := \ln(1 - f) \) and using \( q := 1 - p \) we get the following:

\[
p < 1 - \exp(\sigma \cdot c)
\]

\[
\Leftrightarrow \quad \exp(\sigma \cdot c) < 1 - p \quad (c \text{ is negative})
\]

\[
\Leftrightarrow \quad \frac{1}{\exp(-\sigma \cdot c)} < q
\]

\[
\Leftrightarrow \quad \frac{1}{q} < \exp(-\sigma \cdot c)
\]

The term \( \exp(-\sigma \cdot c) \) can be computed using the Taylor series as described in Section 4.1.

As the relevant information is not the result of the calculation, but whether the given \( p \) is less than or greater than this value, one can also optimize further. Using the Lagrange remainder to estimate the error, it is possible to only compute as many iterations as necessary to get the desired information.

\[
T \exp(x; a) := \sum_{n=0}^{\infty} \frac{x^n}{n!} = \sum_{n=0}^{k} \frac{x^n}{n!} + R_k(x)
\]

Where using an upper bound \( M \) on the domain of \( x \), the remainder (or error term) \( R_k(x) \) can be estimated as follows:

\[
R_k(x) \leq |M \cdot \frac{x^{k+1}}{(k+1)!}|
\]

For the use-case of the leader election, a good integral bound for the error estimation is the maximal value of \( |\ln(1 - f)| \leq M \) for \( f \in [0, 0.95] \) is \( M = 3 \). For any larger value of \( f \), a different value would have to be chosen. The current value for \( f \) is 0.1.

In general, for this use case one can easily estimate \( M \), as the smallest integer larger than the exponential function evaluated at the upper bound of the interval of the domain of \( x \). For each partial sum \( \Sigma_k \), one can then test whether \( p \) is greater than \( \Sigma_k + |R_k(x)| \) or less than \( \Sigma_k - |R_k(x)| \) to decide whether one can stop at an early iteration.

Figure 5 shows the benchmark results for test data using the Haskell implementation. The lower (red) results show the run-time (in \( \mu s \)) for the naive, full computation. The middle (blue) part shows the run-time using a partial pre-computation of \( \ln(1 - f) \) for the exponentiation. The upper (yellow) part shows the run-time using the proposed optimization. For the 10 data points, the first 5 succeed in the leader election, the remaining 5 do not.

References

IOHK Formal Methods Team. A formal specification of the cardano ledger. TODO.
Figure 1: Relative Error for Taylor Series Approximation of ln
Figure 2: Relative Error for Continued Fraction Approximation of ln
Figure 3: Relative Error for Taylor Series Approximation of $\exp$
Figure 4: Relative Error for Continued Fraction Approximation of $\exp$
Figure 5: Benchmark Results for the Haskell Implementation