A Specification of the Non-Integral Calculations in the Shelley Ledger

Deliverable SL-D1

Matthias Güdemann <matthias.gudemann@iohk.io>

Project: Shelley Ledger

Type: Deliverable
Due Date: 20th September 2019

Responsible team: Formal Methods Team
Editor: Matthias Güdemann, IOHK
Team Leader: Philipp Kant, IOHK

Version 0.2
18th September 2019

Dissemination Level		
PU	Public	
CO	Confidential, only for company distribution	
DR	Draft, not for general circulation	$\surd$

Contents

List of Figures

Change Log

Rev.	Date	Who	Team	What
1	05/09/19	Matthias Güdemann	FM (IOHK)	Initial version (0.1).
2	05/09/19	Kevin Hammond	FM (IOHK)	Reviewed and applied comments.
3	12/09/19	Kevin Hammond	FM (IOHK)	Added cover page for consistency.
4	12/09/19	Kevin Hammond	FM (IOHK)	Included review comments from MG.
5	17/09/19	Matthias Güdemann	FM (IOHK)	Reviewed and commented.
6	18/09/19	Kevin Hammond	FM (IOHK)	MG comments incorporated, added deliverable number, frozen to V0.2

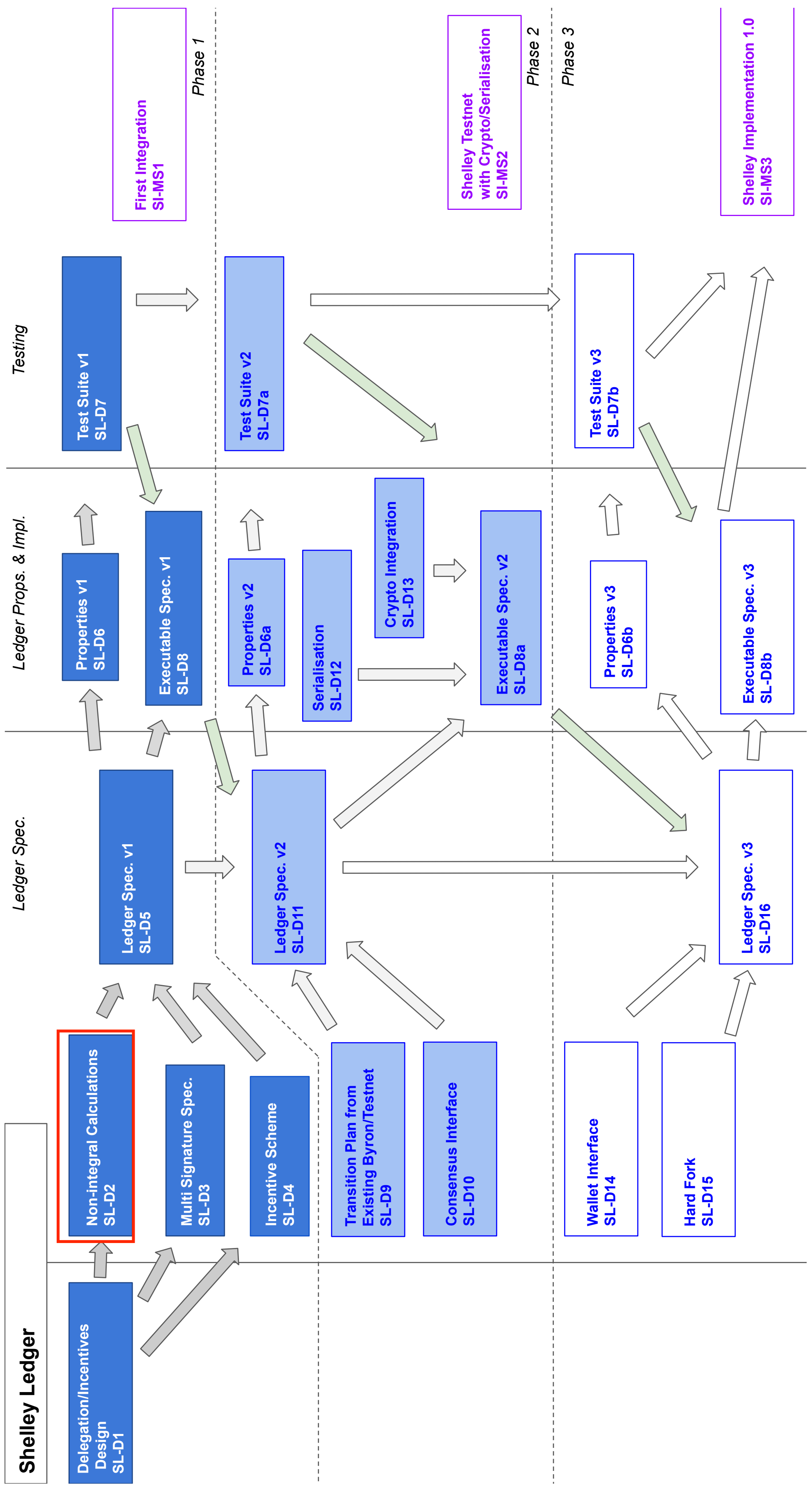

Figure 1: Positioning of this Deliverable (outlined in red).

A Specification of the Non-Integral Calculations in the Shelley Ledger

Matthias Güdemann
matthias.gudemann@iohk.io

May 5, 2022

Abstract

This document defines a consistent way to perform non-integral calculations in the Shelley ledger for Cardano. These calculations are defined in terms of *elementary* mathematical functions, and are used in a few, but important, places in the ledger calculation. The main objective of this work is to provide a clear and unambiguous specification that will give the same results for each calculation, regardless of the computer architecture or programming language implementation that is chosen. The goal is to prevent blockchain forks because of slight differences in the calculated values that might otherwise come from different implementations of the Shelley ledger specification or Ouroboros protocol, for example.

Contents

List of Figures

1 Introduction

The Shelley ledger specification [SL-D5] and the Ouroboros protocol [Ouroboros-Protocol] use non-integral calculations in a few places. Most of these can be implemented in an exact way using rational numbers and arbitrary precision integers, which are available either directly as programming language constructs (e.g. in Haskell) or via external libraries. [1] In addition, there are some instances where the results of elementary functions cannot be precisely represented as rational numbers, and therefore require a representation that covers a wider range of *real numbers*. These include the exponential function $e^x, x \in \mathbb{R}$, which is used to calculate how refunds decay over time, and general non-integral exponentiation $x^y, x, y \in \mathbb{R}$.

2 Problem Description

The calculations in the Shelley ledger specification and Ouroboros protocol that involve elementary functions are concerned with: 1. the decay of the value that should be refunded from a deposit; and 2. the leader election probability. In all these cases, it is important that all distributed nodes calculate the same values, since otherwise there might be a disagreement about what should be included in the blockchain and forks might then result. This is a particular problem where there could be multiple independent implementations that could deliver inconsistent results. Since Cardano aims to be a cryptocurrency that is fully defined by a precise specification,

[1]The most widely used library is the GNU multi-precision library (GMP)

we need an exact specification of non-integer calculations, allowing different implementations to have the same, correct and verifiable, behavior.

3 Implementation Possibilities for Non-Integer Arithmetic

There are three main different possibilities to implement non-integral calculations. In Haskell, we can use typeclasses or parametric polymorphism to design generic algorithms that can handle different choices. Each implementation choice has its own advantages and disadvantages.

3.1 IEEE 754 Floating Point

A straightforward approach is to use IEEE 754 floating-point arithmetic. This is widely supported by common programming languages and CPU/GPU architectures. The basic arithmetic operations of IEEE 754 floating-point are well specified, and they are usually very efficient due to direct HW implementation. Double-precision IEEE 754 floating-point numbers (*binary64*) provide 53 bits (15.99 decimal digits) of precision. This is slightly less than is needed to represent the required fraction of 1 lovelace: there are $4.5 \cdot 10^{16}$ possible fractions. Moreover, the elementary functions that are required for Cardano are not standardized and there can be subtle differences in different implementations, with results even varying when different compiler settings are used on the same implementation[2]. It is also worth noting that some architectures provide excess precision [3], which can result in greater accuracy, but can also give slight differences in results between implementations. Some programming languages, e.g., D, exploit this by default. Many languages support excess precision, but default to IEEE 754 64 bits. In particular on the AMD64 architecture most language implementations use SSE floating-point arithmetic. A final problem is that IEEE 754 supports different rounding modes[4], which will obviously give different results in some cases. Some programming languages allow the rounding mode to be changed, e.g. C99 onwards, but others do not, e.g., Haskell.

3.2 Rational Numbers

Rational numbers can be implemented on top of exact arbitrary precision integral arithmetic using a pair of numbers to represent a numerator and denominator. Effectively, *gcd* calculations are used to normalize the results after each basic operation. While not all real numbers are rationals, this approach allows for arbitrarily precise approximation of all real numbers. However, arbitrarily long numerators or denominators may be needed to represent irrational numbers to the required precision. These arbitrarily long numerators and denominators are undesirable, since they will incur non-predictable [5] run-time.

3.3 Fixed Point Arithmetic

Fixed point arithmetic uses a fixed size value (typically 32 or 64 bits) to represent non-integer numbers. A fixed scaling factor is used to determine the position of the decimal point and normal integer arithmetic operations are applied for addition, multiplication etc. Like floating point, this has the advantage of allowing a compact number representation, and allows fast implementation even on a system without floating point supprt. It also allows precise arithmetic. However, arbitrarily large numbers cannot be supported, which would make it unsuitable

[2]e.g. maintaining intermediate representations within internal registers *vs.* writing to memory.

[3]The original x87 provided 80 bit floating-point

[4]round to nearest (with even digits breaking ties or ties rounding away from zero), round down, round up, round to zero

[5]potentially ¿ 1s for basic arithmetic operations

for representing currency in Cardano (it might, however, be suitable for e.g. leader election calculations).

3.4 Fixed Precision Arithmetic

An alternative approach is to use fixed precision arithmetic based on exact integers where a proportion of the integer is used as fractional part. The number of fractional digits is fixed to a constant. This is more efficient than using the rational number approach described above, since it is not necessary to normalise values after calculations, arbitrary precision results are not supported, and only a single arbitrarily sized integer needs to be stored. Since it is implemented purely in software on top of exact integer arithmetic, this approach will be less efficient than IEEE 754 floating point. However, it can be made to behave equivalently in different implementations, including ensuring consistent (bounded) timings for each architecture, by providing golden tests to validate consistent esults for different architectures etc.

The basic idea is the following: for n decimal digits of precision, multiply each integral value by 10^n and take this into account in the basic operations. For Cardano, this would mean using $n = 17$ to support the required 17 digits of precision that would track each possible stake fraction. Haskell provides library support for fixed precision numbers and arithmetic.

3.5 Exact Real Numbers

Finally, there has been some research work on exact representations of real numbers using e.g. *continued fractions*, exploiting lazy evaluation to produce the required number of digits of precision (e.g. [Esc96]). All real numbers can be represented, whether rational or irrational. These approaches have not yet been widely used, however, except for research purposes. Implementations are often complicated and expensive, and some algorithms may need to be changed (for example, exact equality may be impossible to determine in a finite time).

3.6 Conclusion

For the Cardano cryptocurrency, the fixed precision arithmetic approach seems to be most suitable. It allows for the desired precision and can be implemented in an equivalent way, because it uses exact integer arithmetic. [6] While the fixed-precision approach is less efficient than IEEE 754 floating point, by using partial pre-computation and other techniques, it will be possible to improve performance considerably in some cases (e.g. when using a constant base for exponentiation): experiments show that improvements of two orders of magnitude are possible.

4 Approximation Algorithms

Other than basic arithmetic (addition, multiplication, subtraction, and division), the only elementary functions that are required by the ledger specification are the exponential function, e^x, and its generalisation to the exponentiation function, x^y. Exponentiation of arbitrary real numbers is calculated using the identity: $x^y = \exp(\ln(x^y)) = \exp(y \cdot \ln(x))$. This means that we need suitable approximation schemes for $\exp(x)$ and $\ln(x)$. There are two main approaches that are used for approximation: Taylor / MacLaurin series and continued fractions. Both require only basic arithmetic operations and allow for iterative approximation, i.e., constructing a sequence of $x_0, x_1, \ldots x_n$ where each successive x_i is a better approximation to the desired value.

[6] There already exist two implementations, Haskell and C.

4.1 Taylor Series

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

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

Most commonly the Taylor series is used at $a = 0$. This is also called the MacLaurin series. It uses a truncated, finite polynomial for the function approximation as follows:

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

Using the above, the exponential function can be approximated using the Taylor series by:

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

The natural logarithm can similarly be approximated by:

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

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:

$$b_0 + \cfrac{a_1}{b_1 + \cfrac{a_2}{b_2 + \cfrac{a_3}{\ddots}}}$$

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

$$\begin{aligned}
A_{-1} &:= 1 \\
A_0 &:= b_0 \\
B_{-1} &:= 0 \\
B_0 &:= 1 \\
A_n &:= b_n \cdot A_{n-1} + a_n \cdot A_{n-2} \\
B_n &:= b_n \cdot B_{n-1} + a_n \cdot B_{n-2}
\end{aligned}$$

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

$$\begin{aligned}
\sigma(a_i) &:= 1 \cdot x, -1 \cdot x, -2 \cdot x, \ldots \\
\sigma(b_i) &:= 1, 2 + x, 3 + x, \ldots
\end{aligned}$$

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

$$\begin{aligned}
\sigma(a_i) &:= x, 1^2 \cdot x, 1^2 \cdot x, 2^2 \cdot x, 2^2 \cdot x, 3^2 \cdot x, 3^2 \cdot x, \ldots \\
\sigma(b_i) &:= 1, 2, 3, \ldots
\end{aligned}$$

4.3 Scaling

Neither Taylor series approximations nor continued fractions 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 obtain 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:

$$\ln(x) = \ln\left(\exp(n) \cdot \frac{x}{\exp(n)}\right) = \ln(\exp(n)) + \ln\left(\frac{x}{\exp(n)}\right) = n + \ln\left(\frac{x}{\exp(n)}\right)$$

This 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

Our experimental results have shown that continued fractions converge faster, in particular for the natural logarithm.

Figure 2 shows the relative approximation error for ln with 10, 20, and 50 iterations using a Taylor series. Figure 3 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 4 shows the relative approximation error of 10 and 20 iterations using a Taylor series approximation. Figure 5 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, we conclude that the convergence speed is higher for continued fractions, in particular for the natural logarithm algorithm. However, timing analysis has shown that due to the simpler calculation per iteration of Taylor series, the exponential function can be approximated more efficiently using that approach. We have therefore decided to use both approaches: Taylor series for exp and continued fractions for ln. In order to decide when the approximation has obtained enough precision, we use the following criterion for two successive approximations x_n, x_{n+1}:

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

Both approximation schemes therefore will give 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 [7]. For the Haskell version, Quickcheck property-based tests are provided that will validate the mathematical consistency of the laws for ln and exp. For both the C and Haskell versions, there exist test programs that read two numbers with 34 decimal digits of precision and calculate x^y. This has been used to validate the two different

[7] https://gmplib.org

implementations for 20.000.000 randomly generated testcases, where x and y have been drawn uniformly from the interval $[0.1; 100.1]$. Our results showed that all calculation were identical for all 34 decimal digits. A key aspect to obtaining equivalent results is to use the same approach to rounding after a division. In particular, in our tests we have used rounding to $-\infty$.

6 Optimisations for Specific Use-Cases

One specific 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 locally every 2s for each potential slot leader. The calculation 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, it is necessary to check whether for a given p, σ and a constant f (at least for one epoch), the following inequality holds:

$$p < 1 - (1 - f)^{\sigma}$$

Since $1 - f$ 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 for each epoch, and use it for every following computation in that epoch.

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

$$\begin{array}{rl} & p < 1 - \exp(\sigma \cdot c) \\ \Leftrightarrow & \exp(\sigma \cdot c) < 1 - p \quad \text{(c is negative)} \\ \Leftrightarrow & \frac{1}{\exp(-\sigma \cdot c)} < q \\ \Leftrightarrow & \frac{1}{q} < \exp(-\sigma \cdot c) \\ \Leftrightarrow & \ln(\frac{1}{q}) < (-\sigma \cdot c) \\ \Leftrightarrow & \ln(q) > \sigma \cdot c \end{array}$$

The terms $\exp(-\sigma \cdot c)$ or $\ln(q)$ can be computed using the Taylor series as described in Section 4.1. By using table lookup on $\ln q$, a quick check can be obtained on whether the comparison is definitely true or not true. If it is not certain, then the full calculation can be performed. By using a log calculation, integer sizes will be reduced, which will also improve the performance of the calculation. Alternatively, since the result of the function falls within known limits, we can use linear lower and upper bounds to quickly eliminate comparisons that will certainly fail, without needing to perform the full calculation.

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

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

By 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 leader election use-case, a good integral bound for the error estimation is the maximal value of $|\ln(1 - f)| \leq M$ for $f \in [0, 0.95]$, that is $M = 3$[8] For larger values of f, a different value of M would need to be chosen. The current value for f in the Shelley ledger is 0.1.

[8]Since f is the active slot coefficient, we will very likely stay at lower values, and $M = 12$ would then be enough for $[0, 0.99]$.

In general, M can easily be estimated as the smallest integer that is larger than the exponential function that is evaluated at the upper bound of the interval of the domain of x. For each partial sum Σ_k, it is then possible to test whether p is greater than $\Sigma_k + |R_k(x)|$ or less than $\Sigma_k - |R_k(x)|$ in order to decide whether to stop the calculation at an early iteration.

Figure 6 shows the benchmark results for test data using the Haskell implementation. The lower (red) results show the run-time (in μs) 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

[Esc96] M. H. Escardó. PCF extended with real numbers. *Theor. Comput. Sci.*, 162(1):79–115, 1996. doi:10.1016/0304-3975(95)00250-2.

[Ouroboros-Protocol] A. Kiayias, A. Russell, B. David, and R. Oliynykov. Ouroboros: A provably secure proof-of-stake blockchain protocol. Cryptology ePrint Archive, Report 2016/889, 2016. `http://eprint.iacr.org/2016/889`.

[SL-D5] IOHK Formal Methods Team. A Formal Specification of the Cardano Ledger, IOHK Deliverable SL-D5, 2019. URL `https://github.com/input-output-hk/cardano-ledger/tree/master/eras/shelley/formal-spec/ledger-spec.tex`.

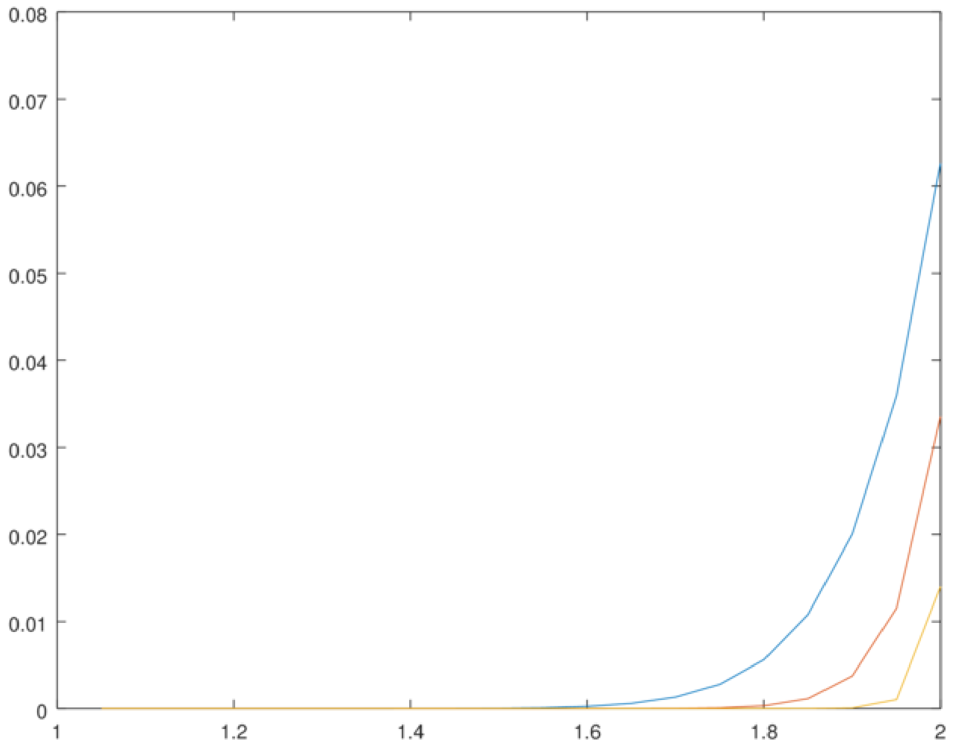

Figure 2: Relative Error for Taylor Series Approximation of ln

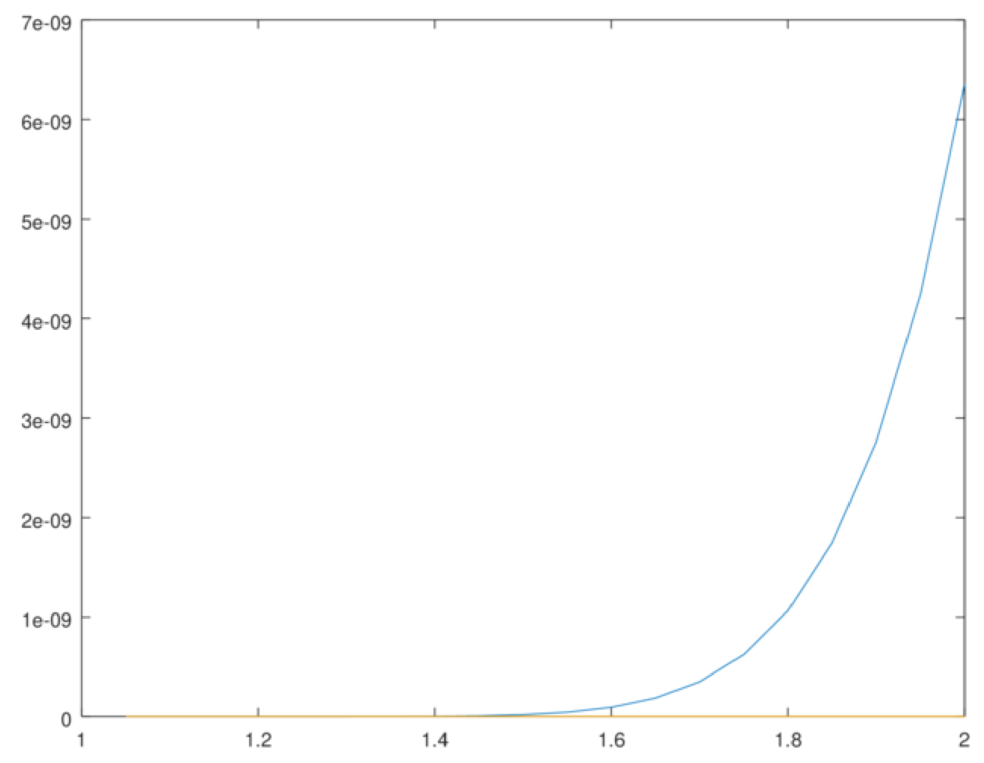

Figure 3: Relative Error for Continued Fraction Approximation of ln

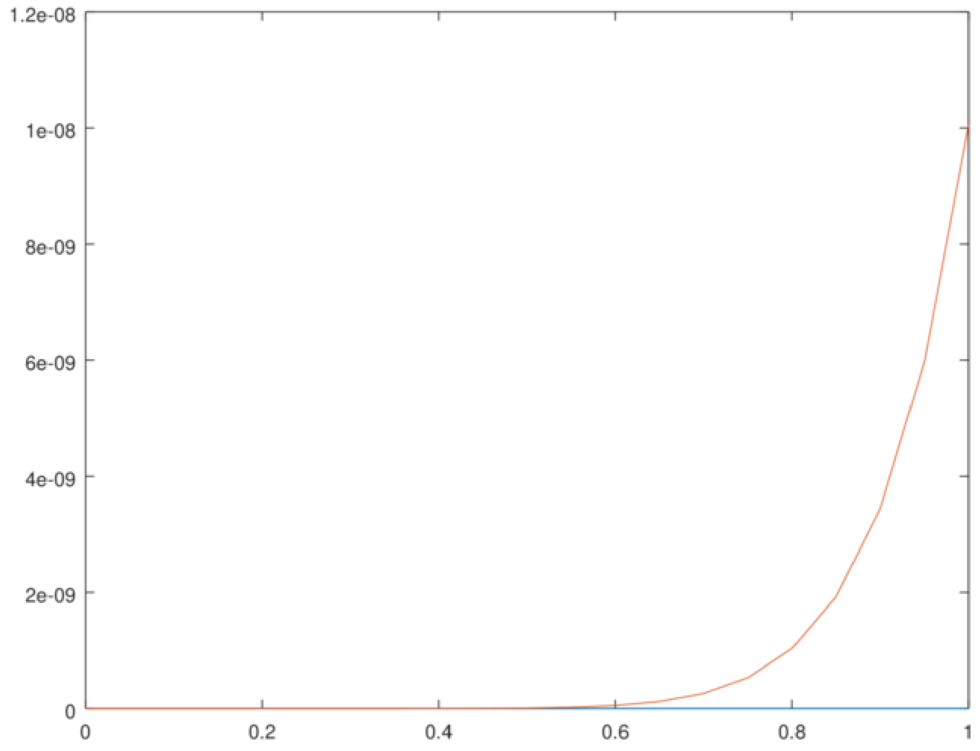

Figure 4: Relative Error for Taylor Series Approximation of exp

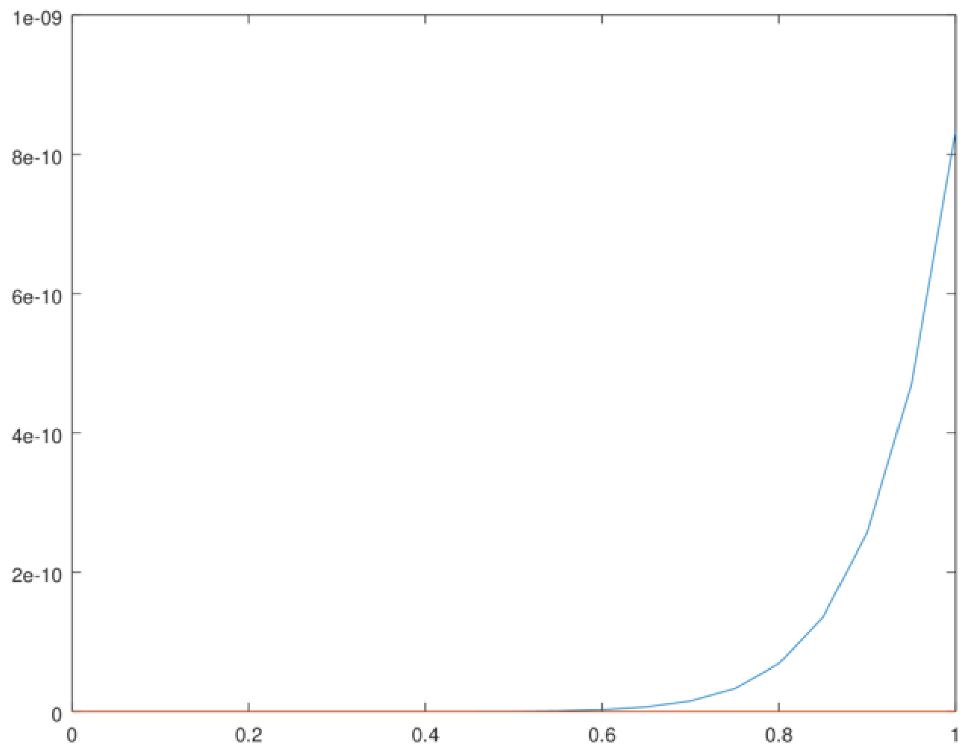

Figure 5: Relative Error for Continued Fraction Approximation of exp

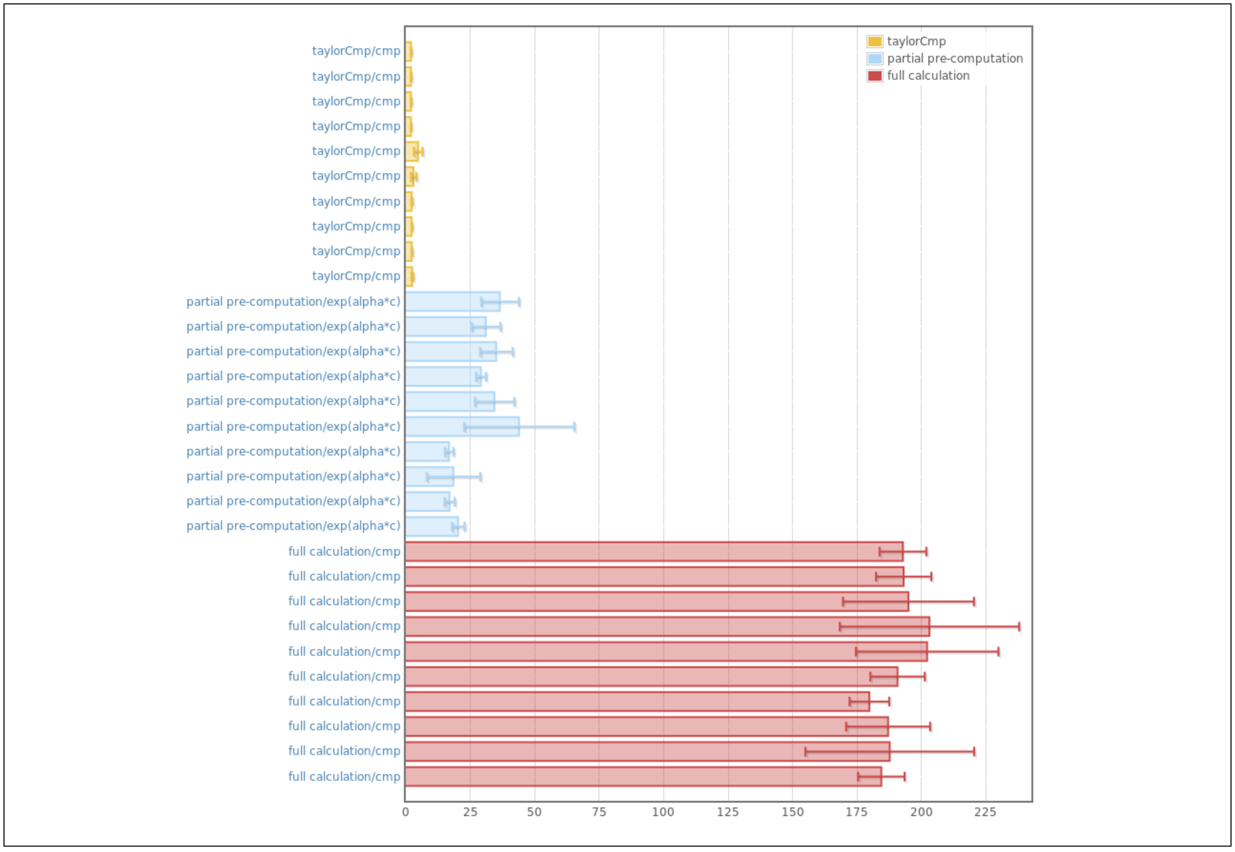

Figure 6: Benchmark Results for the Haskell Implementation