Electronic Preprint for Journal of Information Processing Vol.25

Regular Paper

Speeding up Exact Real Arithmetic on Fast Binary Cauchy Sequences by Using Memoization Based on Quantized Precision

Hideyuki Kawabata[1,a)]

Received: September 26, 2016, Accepted: December 27, 2016

Abstract: Exact Real Arithmetic on Fast Binary Cauchy Sequences (FBCSs) provides us a simple and fairly fast way to obtain numerical results of arbitrary precision. The arithmetic on FBCSs can be implemented concisely in a lazy functional language with unlimited-length integer arithmetic, such that each FBCS is represented by a function that generates approximated values with respect to requested precisions. However, application of the arithmetic on FBCSs to programs such as matrix computations, that usually involve large amount of references to common subexpressions, requires care to avoid the blowup of the amount of computation caused by the fact that approximated values are not shared among multiple references. Although simple memoization might alleviate the situation, the effect would be limited since required precisions for subexpressions tend to be various. In this paper, we present an extended design of the arithmetic on FBCSs that enables the memoization based on *quantized precision*, that is expected to enlarge the reuse rate and reduce the amount of computation without sacrificing the properties of the arithmetic to be exact arithmetic. Numerical experiments by using our prototype libraries in Haskell demonstrated that our approach possesses the potential to outperform existing implementations by orders of magnitude in speed and memory consumption.

Keywords: Fast Binary Cauchy Sequence, exact real arithmetic, lazy evaluation, memoization, Haskell library

1. Introduction

Programming for numerical computation involves a kind of peculiar aspect — you always have to pay much attention to the *accuracy* of the resultant values, even though those values are of basic types of modern programming languages such as Haskell. Ordinary floating-point numbers, standardized and used for decades, can not be unreservedly reliable and situations where catastrophic results are obtained are not unusual. Here we show a plain example taken from Ref. [12]. For a linear system $Ax = b$, where

$$A = \left(a_{ij}\right) = \begin{pmatrix} 64919121 & -159018721 \\ 41869520.5 & -102558961 \end{pmatrix}, \quad b = \begin{pmatrix} 1 \\ 0 \end{pmatrix},$$

solutions are obtained as follows.

$$x_1 = a_{22}/(a_{11}a_{22} - a_{12}a_{21}) = 205117922 \tag{1}$$

$$x_2 = -a_{21}/(a_{11}a_{22} - a_{12}a_{21}) = 83739041 \tag{2}$$

However, by using floating-point operations, totally incorrect results are obtained as $\tilde{x_1} = 102558961.0$ and $\tilde{x_2} = 41869520.5$. Even for expressions as simple as Eqs. (1) and (2), floating-point arithmetic produces strange-looking answers.

Exact real arithmetic offers a simple way of programming to obtain arbitrarily accurate answers for computable real numbers [7]. It is different from variable precision arithmetic, since the user does not have to consider the precisions of intermediate results when writing programs. It is also different from simple verified computation based on a sort of interval arithmetic. By using exact arithmetic, numerical results are not only verified but also as precise as the user demands.

Several ways for implementing exact real arithmetic have been studied [8]. Exact arithmetic on *Fast Binary Cauchy Sequence* (*FBCS*) is one of them [7]. FBCS is a kind of *effective Cauchy sequence* that represents a real number by an infinite sequence of rational values whose elements are approximations of the real value at different levels of accuracy. The precision of each approximate value in an effective Cauchy sequence is defined by the index of the element in the sequence. For example, when a computational result corresponding to a real number x is represented by an effective Cauchy sequence $\{q_0, q_1, q_2, \ldots, q_i, \ldots\}$,

$$k \geq e(p) \Longrightarrow |x - q_k| < \frac{1}{2^p}$$

holds for a recursive function $e : \mathbb{N} \to \mathbb{N}$. From the property, an approximation of x which is accurate up to the absolute error of $1/2^p$ can be obtained by evaluating q_k where $k \geq e(p)$. FBCS is, in a word, a restricted but computationally efficient version of effective Cauchy sequence whose elements are integers. When $\{n_0, n_1, \ldots, n_p, \ldots\}$ is an FBCS, it is representable as a function $f = \backslash p \to n_p$ in a program and the approximation at the precision of p is obtained by evaluating $(f\ p)/2^p$. As will be described in Sections 2 and 3, the arithmetic on FBCSs can be concisely designed [7] and implemented by using a lazy functional language with long integers [9], [16], although other languages such as Java can be used to implement [3]. Thanks to the functionality of pro-

[1] Hiroshima City University, Hiroshima 731–3194, Japan
[a)] kawabata@hiroshima-cu.ac.jp

gramming languages such as overloading of operators, the user can write expressions exactly like Eqs. (1) and (2) to obtain sufficiently accurate results without paying attention to what will be done under the hood. Here we show an example written in Haskell that corresponds to the expressions Eqs. (1) and (2), with slight optimization of common-subexpression elimination.

```
let d  = a11 * a22 - a12 * a21   — 1st ref of a22
let x1 = a22 / d                 — 2nd ref of a22
let x2 = - a21 / d
putStrLn $ "x1=" ++ (show x1) ++ ",x2=" ++ (show x2)
```

The arithmetic on FBCSs, defined formally in Ref. [7], however, has a weak point with respect to computational performance. In programs based on FBCSs, all subexpressions are represented as functions. In the case of the above example, all identifiers constituting numerical expressions denote functions of the type representing an FBCS. Thus, multiple references to common identifiers, such as a_{22} and d, cause multiple and independent evaluation of the same functions. Although applying memoization to each function that represents an FBCS might alleviate the situation [9], the effect would be limited because required precision for each subexpression could vary a lot. For example, two evaluations of a_{22} in the above program would require approximations at different precisions (although the detail of the behavior depends on the data supplied to the program). It must be said that applying the arithmetic on FBCSs to those that involve many common subexpressions such as matrix computations and computation on recursively defined sequences requires care because the blowup of the amount of computation as well as the usage of memory is difficult to avoid entirely.

In this paper, we present the idea of *memoization based on quantized precision* for the arithmetic on FBCSs to realize effective memoization that reduces the amount of computation and memory consumption, without sacrificing the properties of the arithmetic to be exact arithmetic.

Our approach is based on the twofold idea; (1) relaxation of constraints on precisions of approximated values used in each arithmetic operator, and (2) memoization done for *normalized* arguments while keeping the property of the exactness of the arithmetic. In order to reduce the variety of required precisions for each arithmetic operator, we make each operator possible to produce an approximation at a required precision from operands at *any precision* that is greater than the limit. The *relaxation* of constraints on precisions at each operator enables the usage of *constrained* precisions in the course of computation — we can restrict the precisions used in the library functions such that, e.g., multiples of 32. In the case, request for precisions of 30, 31, 32 are all *quantized* to 32 [*1].

The effect of the reuse of precomputed higher-precision approximation, also called as caching, has been discussed [5], [14]. However, as we know of, performance optimization of memoization applied to the arithmetic on FBCSs in a *referentially transparent manner* does not exist.

The contributions of the work reported in this paper are summarized as follows:

- We present extended definition of the arithmetic on FBCSs that can be used to reason about computations for approximation at relaxed, less restricted environment. We also show some essential properties that those arithmetic should possess.
- We present the idea of *memoization based on quantized precision* of the arithmetic on FBCSs and give the implementation in Haskell that are extractable from the extended definition of the arithmetic on FBCSs. We show the arithmetic possesses the property to be exact under quantized precision. The definition is referentially transparent.
- We give some results of numerical experiments carried out on some sensitive (difficult to solve by using floating-point arithmetic), and complicated problems that contain huge amount of common subexpressions. Experimental results demonstrated our approach possesses potential to outperform existing implementations by orders of magnitude in speed and memory consumption.

The organization of this paper is as follows. Sections 2 and 3 summarizes the definition of FBCS and the definition of the arithmetic on FBCSs, respectively. In Section 4, we show the extended definition of the arithmetic on FBCSs to relax requirements of precisions for intermediate approximations. In Section 5, we present the idea of quantized precision to enlarge reuse rate of approximated value for the arithmetic on FBCSs. We define the arithmetic and show that they possess essential properties to be exact arithmetic. Section 6 describes several numerical examples demonstrating the effectiveness of our approach. Related work is covered in Section 7. Finally, we conclude with a brief summary in Section 8.

Throughout this paper, we use Haskell [1] with some extra typesetting features to describe algorithms.

2. Fast Binary Cauchy Sequence and the Arithmetic

2.1 FBCS as a Representation for Real Numbers

Effective Cauchy sequences are not only used for theoretical discussions, but also known to be effective for actual numerical computation. *Fast Binary Cauchy Sequence* (FBCS) is a version of effective Cauchy sequence and is known to be effective for both aspect of the computational speed and memory efficiency [7].

Definition 1. *A Fast Binary Cauchy Sequence* (*FBCS*) *denoting a real v is an infinite sequence of integer values $\{n_0, n_1, \ldots, n_p, \ldots\}$ where the following property holds*

$$\left| v - \frac{n_p}{2^p} \right| < \frac{1}{2^p} , \quad (p = 0, 1, \ldots) . \tag{3}$$

Once you get a sequence $x = \{n_0, n_1, \ldots\}$ that holds the property of Eq. (3) for some v, you can obtain an approximated value of v as $n_p/2^p$ at any precision, i.e., with an absolute error up to $1/2^p$ for any natural number p, since

$$\frac{n_p - 1}{2^p} < v < \frac{n_p + 1}{2^p}$$

[*1] In the example program shown in this section, the approximated values at the two references of a_{22} still can not be shared. However, as shown in Section 6, the effectiveness of our approach is experimentally shown for larger problems.

holds for $p = 0, 1, \ldots$.

Here, we can see an FBCS $x = \{n_0, n_1, \ldots, n_p, \ldots\}$ denoting a real v as a mapping from an index to an integer, i.e., $f_x = \backslash p \to n_p$. The expressions over FBCS are the compositions of those functions. In this view, the model of the computable real numbers can be called a functional representation as in Refs. [3], [5]. Hereafter, we use x (an FBCS which is a sequence of integer) and f_x (the corresponding function from integer to integer) interchangeably.

2.2 Programming with FBCS

In order to construct a program to carry out exact computation by using FBCS, you need (1) generating functions of FBCSs from some representations of numbers used as the sources of the computation, and (2) arithmetic operators on FBCSs, i.e., functions that accept FBCSs as operands and return an FBCS. An entire program will be actually a composition of arithmetic operators and generating functions, followed by the process of extraction of approximated numerical results.

Here, we introduce a simple notation for the relation between a real number and an FBCS.

Definition 2. *We write $v \leftarrow\!\prec x$ if x is an FBCS that denotes a real v.*

The arithmetic operators on FBCSs are expected to satisfy the following basic property of arithmetic, i.e., for any FBCSs x and y and reals v and w,

$$v \leftarrow\!\prec x \Longrightarrow -v \leftarrow\!\prec \mathit{neg}\ x$$
$$v \leftarrow\!\prec x \wedge w \leftarrow\!\prec y \Longrightarrow v + w \leftarrow\!\prec \mathit{add}\ x\ y$$
$$v \leftarrow\!\prec x \wedge w \leftarrow\!\prec y \Longrightarrow v \times w \leftarrow\!\prec \mathit{mul}\ x\ y$$
$$v \leftarrow\!\prec x \Longrightarrow 1/v \leftarrow\!\prec \mathit{recip}\ x$$

where *neg*, *add*, *mul*, and *recip* correspond to real operators – (unary minus), +, ×, and reciprocal, respectively.

Once an FBCS x where $v \leftarrow\!\prec x$ for a real v is constructed, for any non-negative integer p, the approximation of v whose absolute error is less than $1/2^p$ is obtained as $(x\ p)/2^p$. In this paper, we call $(x\ p)/2^p$ the *approximation at precision p of v* where $v \leftarrow\!\prec x$. Note that, in this paper, the word "precision" is not used for the indicator of the relative error that corresponds to the number of meaningful digits, but used for the indicator of the absolute error.

3. Exact Arithmetic on FBCSs

In this section, we show the arithmetic on FBCSs. All the definitions of the arithmetic are taken from Ref. [7]. We show them adding our comments to make the following sections easy to read. Existing implementations [9], [16], [17] are based on the same algorithms. Proofs for properties of operators on FBCSs described in this section are found in Ref. [7].

First, just for convenience, we define a type synonym for the type of functions representing indexed integer sequences [*2].

[*2] Practically, we require unlimited length of integers, such as *Integer* type of Haskell, to define arithmetic operators. However, *Int* will be sufficient to denote precisions of subexpressions, thus the definition of the type synonym *ISeq*. Difference between *Int* and *Integer* is not important for the discussion in this paper.

```
type ISeq = Int → Integer
```

An instance f of *ISeq* that corresponds to the sequence $x = \{n_0, n_1, \ldots, n_p, \ldots\}$ is a function $f = \backslash p \to n_p$. An FBCS is represented as a function of type *ISeq*.

3.1 Generating Function from Literals

Integers are converted to FBCSs by applying the following function:

```
set_z   :: Integer → ISeq
set_z n = \p → 2^p n
```

Constants with decimal points are treated as rationals and are converted to FBCSs by dividing numerators by corresponding denominators, both are FBCSs constructed from integers.

Note that for any integer n, $n \leftarrow\!\prec \mathit{set\_z}\ n$.

3.2 Addition

The sum of two FBCSs are defined as follows:

```
add              :: ISeq → ISeq → ISeq
add x1 x2 p      = r
  where n1       = x1 (p + 2)
        n2       = x2 (p + 2)
        r        = ⌊(n1 + n2)/4⌉
```

Note that $\lfloor q \rceil$ denotes an integer obtained by rounding a rational number q.

For any instances of FBCS x_1 and x_2 and reals v_1 and v_2, $v_1 \leftarrow\!\prec x_1 \wedge v_2 \leftarrow\!\prec x_2 \Longrightarrow v_1 + v_2 \leftarrow\!\prec \mathit{add}\ x_1\ x_2$.

It should be noted that the definition says that if you require an approximation of $v_1 + v_2$ at the precision of p, the approximations of v_1 and v_2, both at the precision of *exactly* $p + 2$, must be used.

3.3 Negation

Negation of an FBCS is defined as the elementwise negation of the sequence.

```
neg      :: ISeq → ISeq
neg x1   = \p → - (x1 p)
```

Obviously, for any instance of FBCS x_1 and a real v_1, $v_1 \leftarrow\!\prec x_1 \Longrightarrow -v_1 \leftarrow\!\prec \mathit{neg}\ x_1$.

3.4 Multiplication

The product of two FBCSs is defined as follows:

```
mul              :: ISeq → ISeq → ISeq
mul x1 x2 p      = r
  where s1       = ⌊log2(|x1 0| + 2)⌋ + 3
        s2       = ⌊log2(|x2 0| + 2)⌋ + 3
        n1       = x1 (p + s2)
        n2       = x2 (p + s1)
        r        = ⌊(n1 n2)/2^(p+s1+s2)⌉
```

$\log_2$ in the above definition is a logarithm function for integer.

If $v_1 \leftarrow\!\prec x_1$ and $v_2 \leftarrow\!\prec x_2$, then $v_1 \times v_2 \leftarrow\!\prec \mathit{mul}\ x_1\ x_2$ holds.

Note that, different from the case of *add*, you can not say approximations of v_1 and v_2 at what precisions are actually used to

compute the approximation at a required precision of the product $v_1 \times v_2$ — it depends on the absolute values of v_1 and v_2.

3.5 Reciprocal

Division between two FBCSs can be composed of multiplication and reciprocal. The reciprocal of an FBCS is defined as follows:

$$
\begin{array}{lcl}
\mathit{recip} & :: & \mathit{ISeq} \to \mathit{ISeq} \\
\mathit{recip}\ x_1\ p & = & r \\
\quad \textbf{where}\ s & = & \min\ \{\ s \in N \mid 3 \le |x_1\ s|\ \} \\
\quad\quad n_1 & = & x_1\ (p + 2s + 2) \\
\quad\quad r & = & \lfloor \dfrac{2^{2p+2s+2}}{n_1} \rceil
\end{array}
$$

If $v_1 \leftarrow\!\!\prec x_1$, then $1/v_1 \leftarrow\!\!\prec \mathit{recip}\ x_1$ holds. Here, we suppose $0 \not\leftarrow\!\!\prec x_1$; otherwise, the evaluation of *recip* will not terminate.

In order to compute the approximation of $1/v_1$ at some given precision, sufficiently precise approximation is searched and used. Note that the searching algorithm is arbitrary. The Exact Real package [9] uses a kind of binary search applicable to monotonic functions.

3.6 Square Root

Square root function is as follows:

$$
\begin{array}{lcl}
\mathit{sqrt} & :: & \mathit{ISeq} \to \mathit{ISeq} \\
\mathit{sqrt}\ x_1\ p & = & r \\
\quad \textbf{where}\ n_1 & = & x_1\ (2p) \\
\quad\quad r & = & \mathit{isqrt}\ n_1
\end{array}
$$

Note that *isqrt* in the above definition is a square root function for integer with rounding toward zero.

If $v_1 \leftarrow\!\!\prec x_1$ and $v_1 \ge 0$, then $\sqrt{v_1} \leftarrow\!\!\prec \mathit{sqrt}\ x_1$ holds.

3.7 Other Functions

Other functions such as trigonometric, logarithmic, and exponential functions are constructed by *power_series* function [7] with appropriate range reduction.

4. Extended Definition of Exact Arithmetic on FBCSs

In this section, we present our extended design of the arithmetic on FBCSs. The arithmetic operators defined in this section are *relaxed* version of the original operators shown in Section 3 in the sense that one can obtain an approximation at any precision *with operands at arbitrarily higher precision than the lower limit* by using the operators.

Here we explain the effect of the modification. Let us consider the situation where you need an approximated sum of two numbers at the precision of p. If you use *add* in Section 3.2, you will be required to calculate approximations of operands at the precision of exactly $p + 2$. If you use *add_a* defined in this section instead, you will just have to prepare approximations of operands at any precision greater than or equal to $p + 2$. Thus, if you happen to have an approximation of the operand at the precision of $p + m + 2$ for some $m \ge 0$, you can reuse it. The chance of reuse might be bigger with *add_a* than with *add*.

We make use of the flexibility of the operators defined in this section to introduce the idea of memoization based on quantized precision and to extract the implementations of our arithmetic library in Haskell, which will be described in Section 5.

Definition 3 (Addition)**.**

$$
\begin{array}{lcl}
\mathit{add\_a} & :: & \mathit{ISeq} \to \mathit{ISeq} \to \mathit{Int} \to \mathit{Int} \to \mathit{ISeq} \\
\mathit{add\_a}\ x_1\ x_2\ m\ l\ p & = & r \\
\quad \textbf{\textit{where}} & n_1 & = x_1\ (p + m + 2) \\
& n_2 & = x_2\ (p + l + 2) \\
& r & = \lfloor \dfrac{2^l n_1 + 2^m n_2}{2^{m+l+2}} \rceil
\end{array}
$$

Definition 4 (Multiplication)**.**

$$
\begin{array}{lcl}
\mathit{mul\_a} & :: & \mathit{ISeq} \to \mathit{ISeq} \to \mathit{Int} \to \mathit{Int} \to \mathit{Int} \to \mathit{Int} \to \mathit{ISeq} \\
\mathit{mul\_a}\ x_1\ x_2\ z_1\ z_2\ m\ l\ p & = & r \\
\quad \textbf{\textit{where}} & s_1 & = \lfloor \log_2(|x_1\ z_1| + 2) \rfloor + 3 \\
& s_2 & = \lfloor \log_2(|x_2\ z_2| + 2) \rfloor + 3 \\
& n_1 & = x_1\ (p + s_2 + m) \\
& n_2 & = x_2\ (p + s_1 + l) \\
& r & = \lfloor \dfrac{n_1 n_2}{2^{p+s_1+s_2+m+l}} \rceil
\end{array}
$$

Definition 5 (Reciprocal)**.**

$$
\begin{array}{lcl}
\mathit{recip\_a} & :: & \mathit{ISeq} \to \mathit{Int} \to \mathit{ISeq} \\
\mathit{recip\_a}\ x_1\ m\ p & = & r \\
\quad \textbf{\textit{where}} & s & = \min\ \{\ s \in N \mid 3 \le |x_1\ s|\ \} \\
& n_1 & = x_1\ (p + 2s + 2 + m) \\
& r & = \lfloor \dfrac{2^{2p+2s+2+m}}{n_1} \rceil
\end{array}
$$

We suppose $0 \not\leftarrow\!\!\prec x_1$; otherwise, the computation of *recip_a* will not terminate.

Definition 6 (Square root)**.**

$$
\begin{array}{lcl}
\mathit{sqrt\_a} & :: & \mathit{ISeq} \to \mathit{Int} \to \mathit{ISeq} \\
\mathit{sqrt\_a}\ x_1\ m\ p & = & r \\
\quad \textbf{\textit{where}}\ n_1 & = & x_1\ (2p + m) \\
\quad\quad r & = & \lfloor \mathit{rsqrt}\ \dfrac{n_1}{2^m} \rceil
\end{array}
$$

Note that *rsqrt* in the above definition is a square root function for rational numbers with rounding toward zero.

Lemma 1. *For any integers* $m, l \ge 0$*,*

(1) $v_1 \leftarrow\!\!\prec x_1 \wedge v_2 \leftarrow\!\!\prec x_2 \Longrightarrow v_1 + v_2 \leftarrow\!\!\prec \mathit{add\_a}\ x_1\ x_2\ m\ l$*,*

(2) for any integers z_1 *and* z_2*, if* $4 > z_1, z_2 \ge 0$*,*
$v_1 \leftarrow\!\!\prec x_1 \wedge v_2 \leftarrow\!\!\prec x_2 \Longrightarrow v_1 \times v_2 \leftarrow\!\!\prec \mathit{mul\_a}\ x_1\ x_2\ z_1\ z_2\ m\ l$*,*

(3) $v_1 \leftarrow\!\!\prec x_1 \wedge v_1 \neq 0 \Longrightarrow 1/v_1 \leftarrow\!\!\prec \mathit{recip\_a}\ x_1\ m$*, and*

(4) $v_1 \leftarrow\!\!\prec x_1 \wedge v_1 \ge 0 \Longrightarrow \sqrt{v_1} \leftarrow\!\!\prec \mathit{sqrt\_a}\ x_1\ m$*.*

Proofs of the above properties are given in Appendix A.1.

Claim 1. *There are following correspondences:*

- $\mathit{add}\ x_1\ x_2 = \mathit{add\_a}\ x_1\ x_2\ 0\ 0$*,*
- $\mathit{mul}\ x_1\ x_2 = \mathit{mul\_a}\ x_1\ x_2\ 0\ 0\ 0\ 0$*,*
- $\mathit{recip}\ x_1 = \mathit{recip\_a}\ x_1\ 0$*, and*
- $\mathit{sqrt}\ x_1 = \mathit{sqrt\_a}\ x_1\ 0$*.*

5. Memoization on Quantized Precision

In this section, we present an implementation of the set of arithmetic operators on FBCSs that are designed such that *memoization based on quantized precision* is applicable.

5.1 Quantized Precision

In order to raise the opportunity where requested precisions for approximations coincide, we restrict the precisions for which approximated values are computed. Instead of allowing to request approximate computation of arbitrary precision, we might restrict the precision to be, e.g., multiples of 4. When approximation of precision p is required, we compute approximation of precision p' where $p' = quantize\ p$. We call this mapping *quantization* because the function *quantize* should be a non-decreasing step function.

Here, we can use a simple definition of *quantize* as follows:

```
quantize    :: Int → Int
quantize p = (div (p − 1) step + 1) ∗ step
```

When you use the above definition of *quantize* with *step* of 4,

```
map quantize [0..18]
    = [0,4,4,4,4,8,8,8,8,12,12,12,12,16,16,16,16,20,20].
```

Memoization of the arithmetic functions on FBCSs is done based on *normalized* precision, where we define *normalization* as the mapping from each level of the step (defined by the step function *quantize*) to a natural number. For example, the normalizing function *normalize* corresponding to the above *quantize* should hold the following equality:

```
map normalize [0..18]
    = [0,1,1,1,1,2,2,2,2,3,3,3,3,4,4,4,4,5,5].
```

The reason of the usage of normalization is to reduce both computational and spatial overhead caused by the memoization.

In the course of computation of approximated values, normalized values sometimes have to be converted to corresponding quantized values. We call the mapping *expansion*.

When we use the definition of *quantize* above, *normalize* and *expand* should be defined such that[*3] $expand\ .\ normalize = quantize$, e.g., as follows.

```
normalize, expand :: Int → Int
normalize p       = div (p − 1) step + 1
expand    p       = step ∗ p
```

5.2 FBCS's Property under Quantized Precision

Here, we introduce the relation $\leftarrow\!\!\prec_q$ with a quantizing function q, such that for any real v and a sequence of integers $x' = \{n_0, n_1, \ldots, n_p, \ldots\}$,

$$v \leftarrow\!\!\prec_q x' \iff \left| v - \frac{n_{(q\ p)}}{2^{(q\ p)}} \right| < \frac{1}{2^{(q\ p)}} \text{ for } p = 0, 1, \ldots .$$

Obviously, for any quantizing function q, $v \leftarrow\!\!\prec x \Longrightarrow v \leftarrow\!\!\prec_q x$ holds. In addition, following properties hold.

Lemma 2. *Let us assume q is a quantizing function. Then, for any integers* $m, l \geq 0$,

(*1*) $v_1 \leftarrow\!\!\prec_q x_1 \wedge v_2 \leftarrow\!\!\prec_q x_2 \Longrightarrow v_1 + v_2 \leftarrow\!\!\prec_q add\_a\ x_1\ x_2\ m\ l$,

(*2*) *for any integers* z_1 *and* z_2, *if* $4 > z_1, z_2 \geq 0$, $v_1 \leftarrow\!\!\prec_q x_1 \wedge v_2 \leftarrow\!\!\prec_q x_2 \Longrightarrow v_1 \times v_2 \leftarrow\!\!\prec_q mul\_a\ x_1\ x_2\ z_1\ z_2\ m\ l$,

[*3] In addition, if $quantize\ (2 * (quantize\ p)) = 2 * (quantize\ p)$ holds, the implementation of square root function becomes quite simple as shown in Section 5.4.

```
data Tree a = Tree a (Tree a) (Tree a)

genTree       :: (Int → b) → Tree b
genTree f     = gen 0
where gen ix = Tree (f ix)
                    (gen $ ix∗2 + 1) (gen $ ix∗2 + 2)

findTree                          :: Tree b → Int → b
findTree tree ix                  = f (bits $ ix + 1) tree
  where f []     (Tree v _ _) = v
        f (0:bs) (Tree _ l _) = f bs l
        f (1:bs) (Tree _ _ r) = f bs r
        bits                  =
              tail . reverse . map (‘mod‘2)
                . takeWhile (> 0) . iterate (‘div‘2)

memo = findTree . genTree
```

Fig. 1 Memoizing functions [27], [28].

(*3*) $v_1 \leftarrow\!\!\prec_q x_1 \wedge v_1 \neq 0 \Longrightarrow 1/v_1 \leftarrow\!\!\prec_q recip\_a\ x_1\ m$, *and*

(*4*) $v_1 \leftarrow\!\!\prec_q x_1 \wedge v_1 \geq 0 \Longrightarrow \sqrt{v_1} \leftarrow\!\!\prec_q sqrt\_a\ x_1\ m$.

Proof. Substitution of $p' = q\ p$ for every p in the proof for Lemma 1 in Section A.1 suffices. □

Once a sequence of integer x where $v \leftarrow\!\!\prec_q x$ for a real v under a quantizing function q is constructed, for any non-negative integer p, the approximation of v whose absolute error is less than $1/2^{(q\ p)}$ is obtained as $(x\ (q\ p))/2^{(q\ p)}$.

5.3 Memoizing Function

Whether quantized precision are used or not, ordinary memoization functions can be applied to the arithmetic functions on FBCS. However, since precisions are limited to non-negative integer numbers, memoizing facility could be optimized for the usage[*4]. For example, we can use the one based on infinite-sized prefix binary tree [27], [28]. For example, we can use the functions described in **Fig. 1**.

5.4 The Arithmetic Operators on FBCSs with Memoization based on Quantized Precision

According to the discussion in Section 4, we can define Haskell functions to construct the arithmetic library, i.e., addition, multiplication, reciprocal, and square root functions, as shown in this section. Other functions such as exponential and trigonometric functions can be constructed using those with the help of power series function which is shown in Appendix A.3.

In the definitions of the arithmetic operators in this section, we use memoizing function named *memo*. Any definition of function *memo* of type $(Int \to \tau) \to (Int \to \tau)$ can be used here as long as $memo\ f$ behaves the same as f where f is of type $Int \to \tau$.

In the definitions in this section, we also use functions named *expand*, *normalize*, and *quantize*. All of them are of type $Int \to Int$. *quantize* must be non-descending. We also assume $expand\ .\ normalize = quantize$, and $quantize\ .\ quantize = quantize$. A concrete example of *quantize*, *normalize*, and *expand* is given in Section 5.1.

Note that precisions given to functions such as add_x' and mul_x' as the last operand are always quantized. However, memo-

[*4] The effect of the choice of memoizing facility will be described in part in Section 6.

ization is done based on the corresponding normalized precisions.

Definition 7 (Addition)**.**

$$
\begin{aligned}
&add\_x,\ add\_x' \quad :: \ ISeq \to ISeq \to ISeq\\
&add\_x\ \ x_1\ x_2\ p \ = \ memo\ (add\_x'\ x_1\ x_2\ .\ expand)\\
&\qquad\qquad\qquad\qquad (normalize\ p)\\
&add\_x'\ x_1\ x_2\ p' \ = \ r\\
&\quad \textbf{where} \quad q \ = \ quantize\ (p'+2)\\
&\qquad\qquad n_1 \ = \ x_1\ q\\
&\qquad\qquad n_2 \ = \ x_2\ q\\
&\qquad\qquad r \ = \ \lfloor \frac{n_1+n_2}{2^{q-p'}} \rceil
\end{aligned}
$$

Definition 8 (Multiplication)**.**

$$
\begin{aligned}
&mul\_x,\ mul\_x' \quad :: \ ISeq \to ISeq \to ISeq\\
&mul\_x\ \ x_1\ x_2\ p = \ memo\ (mul\_x'\ x_1\ x_2\ .\ expand)\\
&\qquad\qquad\qquad\qquad (normalize\ p)\\
&mul\_x'\ x_1\ x_2\ p = \ r\\
&\quad \textbf{where} \ s_1 \ = \ \lfloor \log_2(|x_1\ 0|+2) \rfloor + 3\\
&\qquad\qquad s_2 \ = \ \lfloor \log_2(|x_2\ 0|+2) \rfloor + 3\\
&\qquad\qquad q_1 \ = \ quantize\ (p+s_2)\\
&\qquad\qquad q_2 \ = \ quantize\ (p+s_1)\\
&\qquad\qquad n_1 \ = \ x_1\ q_1\\
&\qquad\qquad n_2 \ = \ x_2\ q_2\\
&\qquad\qquad r \ = \ \lfloor \frac{n_1 n_2}{2^{q_1+q_2-p}} \rceil
\end{aligned}
$$

Definition 9 (Reciprocal)**.**

$$
\begin{aligned}
&recip\_x\ ,\ \ recip\_x'\ \ :: \ ISeq \to ISeq\\
&recip\_x\ \ x_1\ p \quad = \ memo\ (recip\_x'\ x_1\ .\ expand)\\
&\qquad\qquad\qquad\qquad (normalize\ p)\\
&recip\_x'\ \ x_1\ p \quad = \ r\\
&\quad \textbf{where} \ \ s_1 \ = \ head\ \$\ filter\ ((3 \le)\ .\ abs\ .\ x_1)\\
&\qquad\qquad\qquad\qquad \$\ map\ expand\ [0..]\\
&\qquad\qquad q \ = \ quantize\ (p+2s_1+2)\\
&\qquad\qquad n_1 \ = \ x_1\ q\\
&\qquad\qquad r \ = \ \lfloor \frac{2^{q+p}}{n_1} \rceil
\end{aligned}
$$

Computation of s_1 in the definition of *recip_x'*, that is a search for the minimum value in a set, can be done using a kind of binary search used in Ref. [9]. It is difficult to decide the best way, though, because the performance might be significantly affected depending on the application programs.

Definition 10 (Square Root)**.**

$$
\begin{aligned}
&sqrt\_x \quad :: \ ISeq \to ISeq\\
&sqrt\_x \ \ x_1\ p \ = \ memo\ (sqrt\_x'\ x_1\ .\ expand)\\
&\qquad\qquad\qquad\qquad (normalize\ p)\\
&sqrt\_x'\ x_1\ p \ = \ r\\
&\quad \textbf{where} \ n_1 \ = \ x_1\ (2p)\\
&\qquad\qquad r \ = \ isqrt\ n_1
\end{aligned}
$$

isqrt in *sqrt_x'* is a square root function of type *Integer→Integer*, with rounding toward zero. Note that *sqrt_x'* assumes that $quantize\ (2 * (quantize\ p)) = 2 * (quantize\ p)$ for any $q \ge 0$.

5.5 Properties of the Arithmetic

From the following theorem, we can say that the arithmetic operators defined in this section possess the essential properties to be exact arithmetic.

Theorem 1. *For any reals v_1 and v_2, integer sequences x_1 and x_2, the following relations hold under the assumption that appropriate quantizing function is used consistently:*

(1) $v_1 \leftarrow\!\!\prec_q x_1 \wedge v_2 \leftarrow\!\!\prec_q x_2 \Longrightarrow v_1 + v_2 \leftarrow\!\!\prec_q add\_x\ x_1\ x_2$.

(2) $v_1 \leftarrow\!\!\prec_q x_1 \wedge v_2 \leftarrow\!\!\prec_q x_2 \Longrightarrow v_1 \times v_2 \leftarrow\!\!\prec_q mul\_x\ x_1\ x_2$.

(3) $v_1 \leftarrow\!\!\prec_q x_1 \wedge v_1 \neq 0 \Longrightarrow 1/v_1 \leftarrow\!\!\prec_q inv\_x\ x_1$, *and*

(4) *if* $quantize\ (2*(quantize\ p)) = 2*(quantize\ p)$ *for any* $p \ge 0$, $v_1 \leftarrow\!\!\prec_q x_1 \wedge v_1 \ge 0 \Longrightarrow \sqrt{v_1} \leftarrow\!\!\prec_q sqrt\_x\ x_1$.

Proof. (1) Let $q = quantize\ (p'+2)$ and $l = m = q - p' - 2$ for a $p' \ge 0$. Since *quantize* is non-decreasing, $l, m \ge 0$ holds. Then, from the definitions of *add_a* and *add_x'*, $add\_a\ x_1\ x_2\ m\ l\ p'$ is reduced to $add\_x'\ x_1\ x_2\ p'$. With these and Lemma 2, $v_1 + v_2 \leftarrow\!\!\prec_q add\_x'\ x_1\ x_2$.
Now, from the definition of *add_x*, $add\_x\ x_1\ x_2\ (quantize\ p)$ evaluates to the same value as $add\_x'\ x_1\ x_2\ (quantize\ p)$ since the premise of the property of *memo* and the relations $expand\ .\ normalize = quantize$ and $quantize\ .\ quantize = quantize$. This concludes the proof.

(2) Let $s_1 = \lfloor \log_2(|x_1\ 0|+2) \rfloor + 3$, $s_2 = \lfloor \log_2(|x_2\ 0|+2) \rfloor + 3$, $q_1 = quantize\ (p'+s_2)$, $q_2 = quantize\ (p'+s_1)$, $z_1 = z_2 = 0, m = q_1 - p' - s_2$, and $l = q_2 - p' - s_1$, for a $p' \ge 0$. Since $s_1, s_2 \ge 4$, $l, m \ge 0$ hold. Then, from the definitions of *mul_a* and *mul_x'*, $mul\_a\ x_1\ x_2\ z_1\ z_2\ m\ l\ p'$ is reduced to $mul\_x'\ x_1\ x_2\ p'$. With these and Lemma 2, $v_1 \times v_2 \leftarrow\!\!\prec_q mul\_x'\ x_1\ x_2$. Now, similar to the proof for (1), $mul\_x\ x_1\ x_2\ (quantize\ p)$ is shown to evaluate to the same value as $mul\_x'\ x_1\ x_2\ (quantize\ p)$.

(3) Let $s_1 = \min\{s \in N | 3 \le |x_1\ s|\}$, $q = quantize\ (p'+2s_1+2)$, $m = q - p' - 2s_1 - 2$, for a $p' \ge 0$. The rest is similar to the previous.

(4) Let $q = quantize\ p'$, for a $p' \ge 0$, and $m = 0$. The rest is similar.

□

We have to say that the effects of *memoization* itself are difficult to formulate. For example, it would be hard to find out the relation between the used step size of *quantize* and the reduced amount of memory usage because the detailed behavior of the arithmetic operations depends on the values given to the program. However, as shown in Section 6, memoization based on quantized precision appears to have some potential to enhance performance of exact arithmetic on FBCSs.

6. Numerical Experiments

In this section, we present the results of some numerical experiments carried out to evaluate the effectiveness of our approach, i.e., memoization based on quantized precision applied to the arithmetic on FBCSs.

All programs were written in Haskell and compiled using GHC 7.10.3, and were run on a MacBook Pro (Intel Core i7 3.1GHz with 16GB memory) running OS X 10.11.6.

Comparisons were done among following arithmetic libraries on FBCSs:

(1) The Exact-Real Package in Hackage [9] of version 0.12.1. This package uses the module *Data.Function.Memoize* to

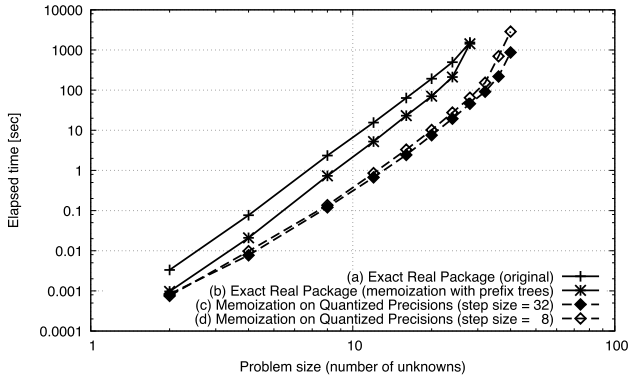

Fig. 2 Elapsed times for solving Hilbert systems by LU factorization. Depicted lines are the results of (a) ER-ORG, (b) ER-MEM, (c) ER-MQR with step size 32, and (d) ER-MQR with step size 8.

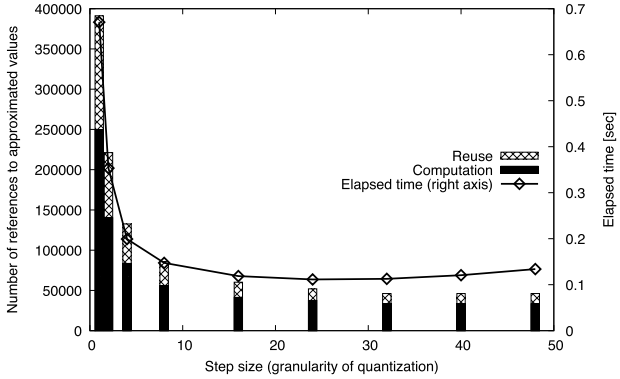

Fig. 3 The relation between the step size and the behavior of the exact arithmetic libraries. Hilbert systems with 8 unknowns were solved. The x-axis is the size of the step for quantization. The bars at step 1, 2, 4, 8, 16, 24, 32, 40, and 48 show the total number of references for approximated values during the computation with the designated step size. The black part and the meshed part show the number of computation for approximations and the number of reuse of precomputed approximations, respectively. The dotted line shows the elapsed time (in seconds, right axis) for each configurations.

memoize the arithmetic functions on FBCSs in a simple way. We consider this is the baseline.

(2) The Exact-Real Package in Hackage that was modified to use a memoizing function shown in Section 5.3.

(3) The exact real arithmetic library on FBCSs constructed with the technique of *memoization based on quantized precision* described in Section 5.4.

In the following, we refer to the arithmetic libraries mentioned above as ER-ORG, ER-MEM, and ER-MQR, respectively. For ER-MQR, the used functions for quantization were those given in Section 5.1.

All the numerical computation were done so that the absolute error of the results were guaranteed to be less than 2^{-53}.

6.1 Experiment 1: Solving Sensitive Linear Systems of Equations

The Hilbert matrix is a square matrix $\left(h_{ij}\right)$, where $h_{ij} = 1/(i + j - 1)$. A linear system of equations with a Hilbert matrix as its coefficient is known to be difficult to solve precisely. Here, we call the linear systems of equations *Hilbert systems*. We solved Hilbert systems of several sizes. We used LU factorization algorithm without pivoting. Right-hand side vectors were set to $(1\ 0\ \ldots\ 0)^T$. We used mutable arrays offered by *Data.Array.IO* for manipulating matrices and vectors.

The results are shown in **Fig. 2**. Memoization based on quantized precision ((c) and (d)) greatly affected the speeding-up of the programs compared to the others, (a) and (b). In the figure, we can only see a slight difference between (c) and (d). However, in general, the performance of ER-MQP would vary depending on used step size. Note that (b) is virtually the same as ER-MQP with step size 1.

In Fig. 2, we can see a clear difference between (a) and (b). We see that different memoizing facility could largely affect the computational speed of the arithmetic on FBCSs, although we do not touch the point any further here.

We should note that other libraries such as iRRAM [23] and IFN [10] might be a few orders of magnitude faster than those examined in this paper.

6.1.1 On the Amount of Computation for Approximate Values and the Reuse Rate

The amount of computation for approximate values and the reuse rate were measured for solving Hilbert systems. The results are shown in **Fig. 3**.

We can see some interesting facts from Fig. 3. The bars in the figure show a rapid decrease of the number of references to approximated values as the step size increases. The trend must have largely affected the reduction of elapsed time as the step size get large. However, the ratio of the number of reused approximated values to that of computed values was almost the same among the variety of step sizes. This could be attributed to the property of ER-MQR, i.e., increased opportunity of reuse by the memoization on quantized precision would cause the reduction of the amount of references to approximated values. When the amount of reuse and computation are both reduced, the ratio of reuse to computation might be less important compared to the amount of references itself.

As shown in Fig. 3, the number of reuse and computation were the same for the execution of step size 32, 40, 48. For larger step size than the point where the reduction of the amount of references were saturated, computation of ER-MQP must have been done at excessively high precision.

Here we put comments on another effect of our arithmetic libraries; memoized or not, usage of the quantized-precision-based reciprocal, i.e., the *recip_x* operator, was effective to reduce the amount of references of approximated values since the number of searched elements of FBCSs was decreased.

Table 1 Memory usage (in megabytes) for solving Hilbert systems with 8 unknowns. Measured by using the GHC's runtime system. For ER-MQP, columns labeled as s1, s8, and s32 are the results with step size 1, 8, and 32, respectively.

	ER-MEM	ER-MQP		
		s1	s8	s32
Maximum Instantaneous Usage	24.5	22.7	4.62	2.17
Total Allocation	624	836	160	118

6.1.2 On the Amount of Memory Consumption

The amount of memory consumption is measured for selected configurations as shown in **Table 1**. Note that each number in the table includes not only the amount of memory to store approximated values but also all memory consumption for each execution

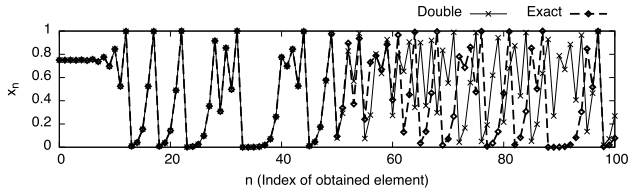

Fig. 4 Values of x_n ($n = 0, \ldots, 100$) of obtained from the computation of the logistic map $x_n = 4x_{n-1}(1-x_{n-1})$ where $x_0 = 0.7501$. Lines show the results of double-precision arithmetic and exact values.

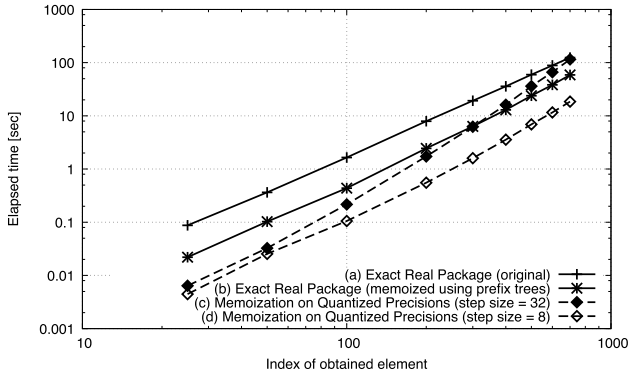

Fig. 5 Elapsed times for computing x_n of the logistic map $x_n = 4x_{n-1}(1 - x_{n-1})$ where $x_0 = 0.7501$. The x-axis corresponds to the index n of x_n. Depicted lines are the results of (a) ER-ORG, (b) ER-MEM, (c) ER-MQR with step size 32, and (d) ER-MQR with step size 8.

of the program.

The table shows that the application of the memoization on quantized precision leads to low memory usage. Roughly speaking, the decreasing trends of the memory usage shown in Table 1 are similar to that depicted by the dotted line in Fig. 3.

6.2 Experiment 2: Computations of Logistic Map

The values of x_n of the logistic map $x_n = 4x_{n-1}(1 - x_{n-1})$ are known to be in the interval $(0, 1)$ and the distribution in the interval is irregular against n. By using double-precision floating-point arithmetic, exact values are difficult to obtain. **Figure 4** shows the plot of x_n for $n = 0, \ldots, 100$. For $n > 50$, the results of floating-point arithmetic is completely different from exact values.

Computational results on the logistic map are summarized in **Fig. 5**. Memoization based on quantized precision (c) was not always better than the original. However, (d) outperformed (b). The difference between (c) and (d) could be explained by looking at **Fig. 6**, where the amount of computation for approximate values and the reuse rate are depicted. From the figure, saturation of the reduction of the amount of references at step size 8 is noticeable. The line (c) in Fig. 5 illustrates the result of the amount of wasted time caused by too precise computations. The amount of memory consumption for selected configurations shown in **Table 2** also confirms the fact.

In Fig. 6, elapsed time increases at higher rate compared to that in Fig. 3 when step size get large. The reason of the difference might be the fact that a simple iterative computation of elements in a sequence, such as the logistic map, constructs a quite unbalanced data-flow graph at runtime. In such a case, overhead caused by the computation at excessively high precision might sensitively affect the computation time. Some optimization found

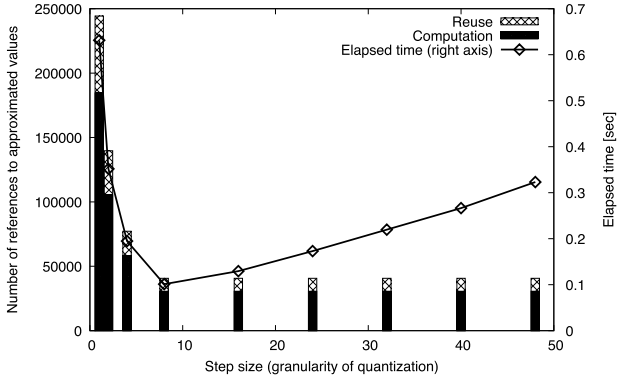

Fig. 6 The relation between the step size and the behavior of the exact arithmetic libraries. x_{100} of the logistic map $x_n = 4x_{n-1}(1 - x_{n-1})$ were computed where $x_0 = 0.7501$. The x-axis is the size of the step for quantization. The bars at step 1, 2, 4, 8, 16, 24, 32, 40, and 48 show the total number of references for approximated values during the computation with the designated step size. The black part and the meshed part show the number of computation for approximations and the number of reuse of precomputed approximations, respectively. The dotted line shows the elapsed time (in seconds, right axis) for each configurations.

Table 2 Memory usage (in megabytes) for the computation of the logistic map. Measured by using the GHC's runtime system. For ER-MQP, columns labeled as s1, s8, and s32 are the results with step size 1, 8, and 32, respectively.

	ER-MEM	ER-MQP		
		s1	s8	s32
Maximum Instantaneous Usage	16.7	26.2	4.44	9.67
Total Allocation	560	756	119	201

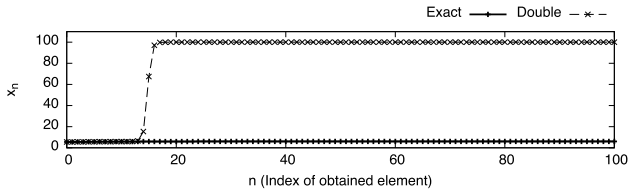

Fig. 7 Values of x_n ($n = 0, \ldots, 100$) of Muller's sequence. The x-axis corresponds to the index n of x_n. Lines show the results of double-precision arithmetic and exact values.

in Refs. [19], [25] could be used to enhance performance for those kind of problems.

6.3 Experiment 3: Muller's Sequences

J.-M. Muller demonstrated the limitation of the floating-point arithmetic using a sequence defined as follows [22]:

$$x_0 = \frac{11}{2}\ ,\quad x_1 = \frac{61}{11}\ ,\quad x_n = 111 - \frac{1130 - (3000/x_{n-2})}{x_{n-1}}.$$

From the specified initial values, $\lim_{n\to\infty} x_n = 6$. However, computational results with double-precision floating-point arithmetic show completely different sequences as shown in **Fig. 7**. Note that 100 and 6 are an attractor and a repellor of the dynamical system defined by the above recurrence relation.

Numerical results on the sequence are shown in **Fig. 8**. **Figure 9** depicts the amount of computation for approximate values and the reuse rate. The amount of memory consumption for selected configurations are shown in **Table 3**. The trends were almost the same as the results described in Section 6.2. The effect of memoization on quantized precision for applying to Muller's sequence was less effective than to the logistic map.

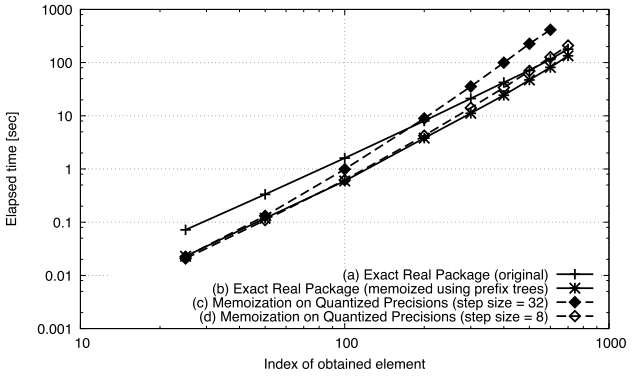

Fig. 8 Elapsed times for computing Muller's sequence. Depicted lines are the results of (a) ER-ORG, (b) ER-MEM, (c) ER-MQR with step size 32, and (d) ER-MQR with step size 8.

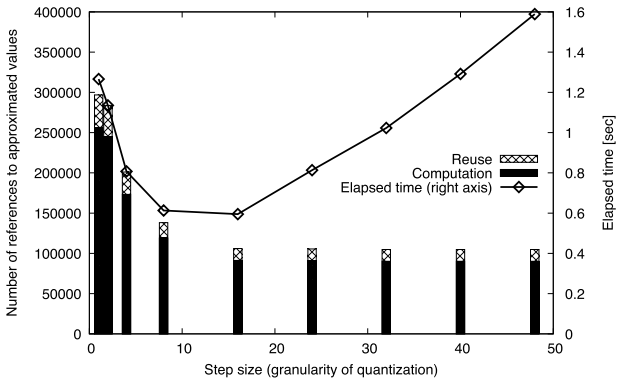

Fig. 9 The relation between the step size and the behavior of the exact arithmetic libraries. Muller's sequence was computed. The x-axis is the size of the step for quantization. The bars at step 1, 2, 4, 8, 16, 24, 32, 40, and 48 show the total number of references for approximated values during the computation with the designated step size. The black part and the meshed part show the number of computation for approximations and the number of reuse of precomputed approximations, respectively. The dotted line shows the elapsed time (in seconds, right axis) for each configurations.

Table 3 Memory usage (in megabytes) for the computation of Muller's sequence. Measured by using the GHC's runtime system. For ER-MQP, columns labeled as s1, s8, and s32 are the results with step size 1, 8, and 32, respectively.

	ER-MEM	ER-MQP		
		s1	s8	s32
Maximum Instantaneous Usage	34.3	63.1	25.3	43.0
Total Allocation	578	1168	583	745

7. Related Work

The idea of exact real arithmetic is directly connected to the study of computable real numbers that has been long studied [11], [15], [21], [24], [26]. There are several definitions of computable reals [8], [15], [24]. Among them, Cauchy sequence can be used to represent numbers in a class of computable real numbers, i.e., recursive real numbers [24], and is known to be the one which best expresses the existence of an algorithm permitting one to calculate uniformly the terms of a sequence with any desired degree of accuracy [21]. Fast Binary Cauchy Sequence (FBCS), that is a modified version of effective Cauchy sequence, was designed to be efficient with respect to the computational speed and memory consumption [7].

Performance issues of exact arithmetic has been studied [4], [14]. Reference [4] reports that the approach to represent real numbers by infinite digits might not be attractive with respect to computational speed. Optimization at several levels had been proposed; supplying multiple levels of interfaces [13], and applying static analysis to avoid too precise computation while performing top-down error propagation [19], [25]. For FBCSs, caching of approximated values using mutable memory to reduce the amount of computation has been proposed [3], [5].

The correctness is essential for exact arithmetic libraries to be, indeed, exact arithmetic. Ménissier-Morain first proved the correctness of her Cauchy sequence-based exact arithmetic [20]. FBCS is formally defined and the properties were shown in Ref. [7] and validated by using PVS [17], [18]. Coq is also applied to certify correctness of exact arithmetic (of streams of digits) [6].

In this paper, we proposed a way to enhance the computational efficiency of FBCS in a referentially transparent manner, with proof of the correctness of the extended arithmetic on FBCSs.

8. Conclusion

In this paper, we presented an approach to speed up the computation of exact arithmetic based on Fast Binary Cauchy Sequences — memoization on quantized precision. We introduced an extended definitions for arithmetic operators on FBCSs and presented a Haskell implementation for them, in a referentially transparent way, both with the proof of correctness as an exact arithmetic under quantized precision. With the implementation, we carried out several experiments to show the effectiveness of our approach. Some results exhibited that programs with our libraries run more than ten times faster than with an exact real package available in Hackage.

Future work includes a detailed study on the "proper" quantizing funcion. In the numerical experiments described in Section 6, we used the most simple definition of quantizing function given in Section 5.1. The performance of the library with the memoization on quantized precision might depend on the definition of the quantizing function. The effect of the usage of "non-uniform" step functions as quantizing functions should be examined.

There are applications that do not fit the approach of the memoization on quantized precision. For example, our approach would not improve the performance of the computation of calculator style problems very much, such as calculating $e^{\pi\sqrt{163}}$ [2], unless the problems contain large amount of common subexpressions.

Acknowledgments This work was supported in part by an HCU Grant for Special Academic Research (General Studies) under Grant No.1030301.

References

[1] Marlow, S. (Ed.): Haskell 2010 Language Report (2010).
[2] Blanck, J.: Exact Real Arithmetic Systems: Results of Competition, *Proc. CCA 2000, LNCS 2064*, pp.389–393 (2001).
[3] Boehm, H.-J.: The Constructive Reals as a Java library, *Journal of Logic and Algebraic Programming*, Vol.64, pp.3–11 (2005).
[4] Boehm, H.-J., Cartwright, R., Riggle, M. and O'Donnell, M.J.: Exact Real Arithmetic: A Case Study in Higher Order Programming, *Proc. 1986 ACM Conference on Lisp and Functional Programming*, ACM (1986).
[5] Boehm, H.-J.: Constructive Real Interpretation of Numerical Programs, *Proc. 1987 ACM Conference on Interpreters and Interpretives*

Techniques, ACM (1987).
[6] Ciaffaglione, A. and Gianantonio, P.D.: A certified, corecursive implementation of exact real numbers, *Theoretical Computer Science*, Vol.351, pp.39–51 (2006).
[7] Gowland, P. and Lester, D.: The Correctness of an Implementation of Exact Arithmetic, *Proc. 4th Real Numbers and Computers Conference*, pp.125–140 (2000).
[8] Gowland, P. and Lester, D.: A Survey of Exact Arithmetic Implementations, *Proc. CCA 2000, LNCS 2064*, pp.30–47 (2001).
[9] Hermaszewski, J.: The exact-real package, version 0.12.1, available from ⟨http://hackage.haskell.org/package/exact-real⟩ (accessed 2016-04).
[10] Kawabata, H. and Iwasaki, H.: Improving Floating-Point Numbers: A Lazy Approach to Adaptive Accuracy Refinement for Numerical Computations, *Proc. ESOP 2016, LNCS 9632*, pp.390–418 (2016).
[11] Ko, K.-I.: On the Definitions of Some Complexity Classes of Real Numbers, *Mathematical Systems Theory*, Vol.16, pp.95–109 (1983).
[12] Krämer, W.: A Priori Worst Case Error Bounds for Floating-Point Computations, *IEEE Transactions on Computers*, Vol.47, No.7, pp.750–756 (1998).
[13] Lambov, B.: RealLib: An Efficient Implementation of Exact Real Arithmetic, *Mathematical Structures in Computer Science*, Vol.17, No.1, pp.81–98 (2007).
[14] Lee Jr., V.A. and Boehm, H.-J.: Optimizing Programs over the Constructive Reals, *Proc. PLDI*, pp.102–111 (1990).
[15] Lehman, R.S.: On primitive recursive real numbers, *Fundamenta Mathematicae*, Vol.49, No.2, pp.105–118 (1961).
[16] Lester, D.: ERA: Exact Real Arithmetic, version 1.0, available from ⟨http://hackage.haskell.org/package/numbers-3000.2.0.1/docs/Data-Number-CReal.html⟩ (accessed 2016-04).
[17] Lester, D.: The world's shortest correct exact real arithmetic program?, *Information and Computation*, Vol.216, pp.39–46 (2012).
[18] Lester, D. and Gowland, P.: Using PVS to validate the algorithms of an exact arithmetic, *Theoretical Computer Science*, Vol.291, pp.203–218 (2003).
[19] Li, Y. and Yong, J.-H.: Efficient Exact Arithmetic over Constructive Reals, *The 4th Annual Conference on Theory and Applications of Models of Computation* (2007).
[20] Ménissier-Morain, V.: Arbitrary precision real arithmetic: Design and algorithms, *The Journal of Logic and Algebraic Programming*, Vol.64, pp.13–19 (2005).
[21] Mostwski, A.: On computable sequences, *Fundamenta Mathematicae*, Vol.44, No.1, pp.37–51 (1957).
[22] Muller, J.-M.: *Arithmétique des ordinateurs*, Masson (1989).
[23] Müller, N.T.: The iRRAM: Exact Arithmetic in C++, *Proc. CCA 2000, LNCS 2064*, pp.222–252 (2001).
[24] Rice, H.G.: Recursive Real Numbers, *Proc. American Mathematical Society*, Vol.5, No.5, pp.784–791 (1954).
[25] van der Hoeven, J.: Computations with effective real numbers, *Theoretical Computer Science*, Vol.351, pp.52–60 (2006).
[26] Weihrauch, K. and Kreitz, C.: Representations of the Real Numbers and of the Open Subsets of the set of Real Numbers, *Annals of Pure and Applied Logic*, Vol.35, pp.247–260 (1987).
[27] Tanaka, H.: Another way of implementing memoization in Haskell (in Japanese), available from ⟨http://d.hatena.ne.jp/tanakh/20100411/p1⟩ (accessed 2016-04).
[28] Haskell.org: Memoization, available from ⟨https://wiki.haskell.org/Memoization⟩ (accessed 2016-04).

Appendix

A.1 Proofs of Lemmas in Section 4

The proofs shown here basically use the techniques in Ref. [7].

Lemma 1 (1) (Addition)

Proof. We must show the inequality $(r-1)/2^p < v_1+v_2 < (r+1)/2^p$. First, $v_1 \leftarrow\!\shortmid x_1$ and $v_2 \leftarrow\!\shortmid x_2$ implies $\frac{n_1-1}{2^{p+m+2}}+\frac{n_2-1}{2^{p+l+2}} < v_1+v_2 < \frac{n_1+1}{2^{p+m+2}}+\frac{n_2+1}{2^{p+l+2}}$. Next, $r=\lfloor\frac{2^l n_1+2^m n_2}{2^{m+l+2}}\rceil$ and Lemma 4 implies, for $p>0$, $\frac{r-1}{2^p}+\frac{1}{2^{p+1}} < \frac{2^l n_1+2^m n_2}{2^{p+m+l+2}} < \frac{r+1}{2^p}-\frac{1}{2^{p+1}}$. From these inequalities, for $m,l\geq 0$, we can derive the following.

$$\begin{aligned}\frac{r-1}{2^p} &\leq \frac{r-1}{2^p}+\frac{1}{2^{p+1}}-\frac{2^l+2^m}{2^{p+m+l+2}}\\ &\leq \frac{2^l n_1+2^m n_2}{2^{p+m+l+2}}-\frac{2^l+2^m}{2^{p+m+l+2}} < v_1+v_2\\ v_1+v_2 &< \frac{2^l n_1+2^m n_2}{2^{p+m+l+2}}+\frac{2^l+2^m}{2^{p+m+l+2}}\\ &< \frac{r+1}{2^p}-\frac{1}{2^{p+1}}+\frac{2^l+2^m}{2^{p+m+l+2}} \leq \frac{r+1}{2^p}\end{aligned}$$

□

Lemma 1 (2) (Multiplication)

Proof. We show the inequality $(r-1)/2^p < v_1v_2 < (r+1)/2^p$. Since $v_1 \leftarrow\!\shortmid x_1$ and $v_2 \leftarrow\!\shortmid x_2$, $\frac{n_1-1}{2^{p+s_2+m}} < v_1 < \frac{n_1+1}{2^{p+s_2+m}}$ and $\frac{n_2-1}{2^{p+s_1+l}} < v_2 < \frac{n_2+1}{2^{p+s_1+l}}$ hold, so do the following

$$\begin{cases}\dfrac{n_1n_2-|n_1|-|n_2|-1}{2^{2p+s_1+s_2+m+l}} \leq \dfrac{\min\{(n_1\pm 1)(n_2\pm 1)\}}{2^{2p+s_1+s_2+m+l}} < v_1v_2\ ,\\ v_1v_2 < \dfrac{\max\{(n_1\pm 1)(n_2\pm 1)\}}{2^{2p+s_1+s_2+m+l}} \leq \dfrac{n_1n_2+|n_1|+|n_2|+1}{2^{2p+s_1+s_2+m+l}}.\end{cases}$$

From r and Lemma 4, $\frac{r-1}{2^p}+\frac{1}{2^{p+1}} < \frac{n_1n_2}{2^{2p+s_1+s_2+m+l}} < \frac{r+1}{2^p}-\frac{1}{2^{p+1}}$. So,

$$\begin{aligned}\frac{r-1}{2^p}+\left(\frac{1}{2^{p+1}}-\frac{|n_1|+|n_2|+1}{2^{2p+s_1+s_2+m+l}}\right) &\leq \frac{n_1n_2-|n_1|-|n_2|-1}{2^{2p+s_1+s_2+m+l}}\\ &< v_1v_2\end{aligned}$$

and

$$\begin{aligned}\frac{r+1}{2^p}-\left(\frac{1}{2^{p+1}}-\frac{|n_1|+|n_2|+1}{2^{2p+s_1+s_2+m+l}}\right) &> \frac{n_1n_2+|n_1|+|n_2|+1}{2^{2p+s_1+s_2+m+l}}\\ &> v_1v_2,\end{aligned}$$

and this implies it is enough to show $|n_1|+|n_2|+1 \leq 2^{p+s_1+s_1+m+l-1}$. Now, from Lemma 6, if $p>p'\geq 0$, $|x\ p|+1 < 2^{p-p'}|x\ p'|+2^{p-p'}+2 \leq 2^{p-p'}(|x\ p'|+2)$ holds. So, in our case, from Lemma 5 and the definitions of s_1 and s_2, we get the following.

$$\begin{cases}|n_1|+1 < 2^{p+s_2+m-z_1}(|x_1\ z_1|+2) < 2^{p+s_1+s_2+m-z_1-2}\\ |n_2|+1 < 2^{p+s_1+l-z_2}(|x_2\ z_2|+2) < 2^{p+s_1+s_2+l-z_2-2}\end{cases}$$

Thus, since $l,z_1,m,z_2\geq 0$,

$$|n_1|+|n_2|+1 < 2^{p+s_1+s_2-2}(2^{m-z_1}+2^{l-z_2})-1 < 2^{p+s_1+s_1+m+l-1}\ .$$

□

Lemma 1 (3) (Reciprocal)

Proof. We show $(r-1)/2^p < 1/v_1 < (r+1)/2^p$. Since $v_1 \leftarrow\!\shortmid x_1$, $\frac{n_1-1}{2^{p+2s+2+m}} < v_1 < \frac{n_1+1}{2^{p+2s+2+m}}$. So, because $(n_1-1)(n_1+1)>0$, $\frac{2^{p+2s+2+m}}{n_1+1} < \frac{1}{v_1} < \frac{2^{p+2s+2+m}}{n_1-1}$. From Lemma 6, if $p,p'\geq 0$, $(x\ p'-1)2^{p-p'}-1 < x\ p < (x\ p'+1)2^{p-p'}+1$. So, in our case, $(x_1\ s-1)2^{p+s+2+m}-1 < n_1$, that implies $n_1 > 2^{p+s+3+m}-1 \geq 7$ since $x_1\ s\geq 3$. So, $n_1(n_1-1) > (2^{p+s+3+m}-1)(2^{p+s+3+m}-2) = 2^{2p+2s+3+m}\cdot 8\cdot 2^m - 3\cdot 2^{p+s+3+m}+2$. Thus, since $p,s,m\geq 0$,

$$\begin{aligned}&n_1(n_1-1)-2^{2p+2s+3+m}\\ &\quad= 2^{2p+2s+3+m}(8\cdot 2^m-1)-3\cdot 2^{p+s+3+m}+2\\ &\quad= 2^{p+s+3+m}(2^{p+s}(8\cdot 2^m-1)-3)+2\\ &\quad> 0\end{aligned}$$

and $2^{2p+2s+3+m} < n_1(n_1-1) < n_1(n_1+1)$ follows. Now, from the definition of r and Lemma 4, $r-\frac{1}{2} < \frac{2^{2p+2s+2+m}}{n_1} < r+\frac{1}{2}$. Using

these, the followings are shown.

$$\begin{aligned}(r-1)(n_1+1) &< \left(\frac{2^{2p+2s+2+m}}{n_1} - \frac{1}{2}\right)(n_1+1)\\ &= 2^{2p+2s+2+m} + \frac{2^{2p+2s+2+m}}{n_1} - \frac{1}{2}(n_1+1)\\ &= 2^{2p+2s+2+m} + \frac{1}{2n_1}\left(2^{2p+2s+3+m} - n_1(n_1+1)\right)\\ &< 2^{2p+2s+2+m}\end{aligned}$$

$$\begin{aligned}(r+1)(n_1-1) &> \left(\frac{2^{2p+2s+2+m}}{n_1} + \frac{1}{2}\right)(n_1-1)\\ &= 2^{2p+2s+2+m} - \frac{2^{2p+2s+2+m}}{n_1} + \frac{1}{2}(n_1-1)\\ &= 2^{2p+2s+2+m} - \frac{1}{2n_1}\left(2^{2p+2s+3+m} - n_1(n_1-1)\right)\\ &> 2^{2p+2s+2+m}\end{aligned}$$

Thus,

$$\frac{r-1}{2^p} < \frac{2^{p+2s+2+m}}{n_1+1} < \frac{1}{v_1} < \frac{2^{p+2s+2+m}}{n_1-1} < \frac{r+1}{2^p}\ .$$

□

Lemma 1 (4) (Square Root)

Proof. We must show the inequality $(r-1)/2^p < \sqrt{v_1} < (r+1)/2^p$. r must be greater than or equal to zero. Note that from Lemma 3, $2^m r^2 \le n_1 < 2^m(r+1)^2$, i.e., $n_1 + 1 \le 2^m(r+1)^2$.

- Suppose $r = 0$. Then, $v_1 \leftharpoonup x_1$ and $v_1 \ge 0$ implies $0 \le v_1 < \frac{n_1+1}{2^{2p+m}}$. So, we just have to show $\frac{n_1+1}{2^{2p+m}} \le \frac{(r+1)^2}{2^{2p}}$, that is simplified to be $n_1 + 1 \le 2^m(r+1)^2$.
- Suppose $r \ge 1$. Then, $v_1 \leftharpoonup x_1$ and $v_1 \ge 0$ implies $\frac{n_1-1}{2^{2p+m}} < v_1 < \frac{n_1+1}{2^{2p+m}}$. Now, we have to show $\frac{(r-1)^2}{2^{2p}} \le \frac{n_1-1}{2^{2p+m}}$ and $\frac{n_1+1}{2^{2p+m}} \le \frac{(r+1)^2}{2^{2p}}$, i.e., $2^m(r-1)^2 \le n_1 - 1$ and $n_1 + 1 \le 2^m(r+1)^2$. The former can be shown because $1 \le 2^m(2r-1)$ and $2^m r^2 \le n_1$.

□

A.2 Useful Facts from Ref. [7]

Lemma 3. *For $r \in \mathbb{Z}$ and $n \in \mathbb{R}$, $r = \lfloor \sqrt{n} \rfloor \Longrightarrow r^2 \le n < (r+1)^2$.*

Lemma 4. *For $p, q, n \in \mathbb{Z}$, $q > 0, n = \lfloor p/q \rceil \Longrightarrow (n - (1/2))\,q \le p < (n + (1/2))\,q$.*

Lemma 5. *For $n \in \mathbb{Z}$ and $m \in \mathbb{R}$, $n = \lfloor \log_2(m) \rfloor \Longrightarrow 2^n \le m < 2^{n+1}$.*

Lemma 6. *For integers $p, p' \ge 0$ and FBCS x, the following inequality holds:*

$(n_{p'} - 1)2^{p-p'} - 1 < n_p < (n_{p'} + 1)2^{p-p'} + 1$.

A.3 Implementation of the Basic Mathematical Functions using Power Series

Definition of the basic mathematical functions such as sin, exp, and log can be constructed by using a power series function with appropriate range reduction [7], [9]. The function *powerSeries* in Ref. [9] can be extended to adopt memoization on quantized precision as follows:

```
powerSeries :: [Rational] → (Int → Int) → ISeq → ISeq
powerSeries qs f x p
        = memo (powSer qs f x . expand) (normalize p))
powSer qs f x p =  r /. 4^d
  where t  =  f p
        d  =  log2(toInteger t) + 2
        p' =  p + d
        p'' =  quantize $ p' + d
        d2 =  p'' − (p' + d)
        m  =  x p''
        xs =  (%1) < $> iterate (\e → m ∗ e /. 2^p'') (2^p')
        r  =  sum . take (t + 1) . fmap (round . (∗ (2^d)))
              $ zipWith (∗) qs xs
```

The above definition of *powerSeries* is used to compute the approximation of the sum

$$r = \sum_{i=0}^{\infty} Q_i x^i$$

where Q_i and x are list of rationals and a computable real, respectively, where $|x| < 1$, $|Q_i| \le 1$ for all $i \ge 0$, and $|\sum_{i=t+1}^{\infty} Q_i x^i| < 2^{-(p+1)}$ for some t which is dependent on p. With appropriate f such as $f = \backslash n \to \max\ 1\ n$, the following inequalities hold:

$$\frac{r-1}{2^{p'}} < \mathit{powerSeries}\ Q\ f\ x\ p < \frac{r+1}{2^{p'}}$$

where $p' = p + \lfloor \log_2 (f\ p) \rfloor + 2$. Using the result, for example, the function *sin* can be defined as below:

```
sin          :: ISeq → ISeq
sin    x p = r
  where f = max 1 p
        Q = {1/1!, −1/3!, ..., (−1)^n/(2n+1)!, ...}
        r = powerSeries Q f x p
```

For more details, see Refs. [7] and [9].

Hideyuki Kawabata received his B.E. and Ph.D. degrees from Kyoto University in 1992 and 2004, respectively. Since 2007, he has been a lecturer of Hiroshima City University. His research interest includes numerical programming and programming languages. He is a member of ACM, IEEE Computer Society, IPSJ, IEICE, Japan Society for Industrial and Applied Mathematics (JSIAM), and Japan Society for Software Science and Technology (JSSST).