Improving Floating-Point Numbers: A Lazy Approach to Adaptive Accuracy Refinement for Numerical Computations

Hideyuki Kawabata[1(✉)] and Hideya Iwasaki[2]

[1] Hiroshima City University,
3-4-1 Ozuka-Higashi, Asa-Minami-Ku, Hiroshima 731-3194, Japan
kawabata@hiroshima-cu.ac.jp
[2] The University of Electro-Communications,
1-5-1 Chofugaoka, Chofu, Tokyo 182-8585, Japan
iwasaki@cs.uec.ac.jp

Abstract. Numerical computation using floating-point numbers is prone to accuracy degradation due to round-off errors and cancellation of significant digits. Although multiple-precision arithmetic might alleviate this problem, it is difficult to statically decide the optimal degree of precision for each operation in a program. This paper presents a solution to this problem: the partial results in floating-point representations are incrementally improved using adaptive control. This process of adaptive accuracy refinement is implemented using lazy lists, each one containing a sequence of floating-point numbers with gradually improving accuracies. The computation process is driven by the propagation of demand for more accurate results. The concept of this *improving floating-point number* (IFN) mechanism was experimentally implemented in two ways: as a Haskell library and as a pure C library. Despite the simple approach, the results for numerical problems demonstrated the effectiveness of this mechanism.

Keywords: Improving floating-point numbers · Accurate numerical computation · Lazy evaluation · Haskell library

1 Introduction

Obtaining accurate results from numerical computation is not an easy task. Since real values cannot be represented correctly using a fixed number of digits, ordinary numerical computation is carried out using approximated representations of numbers. Programmers have to be fully aware that computation based on approximated numbers can easily degrade the accuracy of the result unexpectedly.

In *floating-point representation* using base 2, each value is denoted as $s \times m \times 2^e$, where s, m, and e are the *sign*, *mantissa*, and *exponent*, respectively. The mantissa (also called *significant digits*) is particularly important since it affects the precision of each number. There are many cases in which the mantissa of a resultant value has few meaningful digits, or even no digit, due to the accumulation of round-off errors or catastrophic cancellation [1].

P. Thiemann (Ed.): ESOP 2016, LNCS 9632, pp. 390–418, 2016.
DOI: 10.1007/978-3-662-49498-1_16

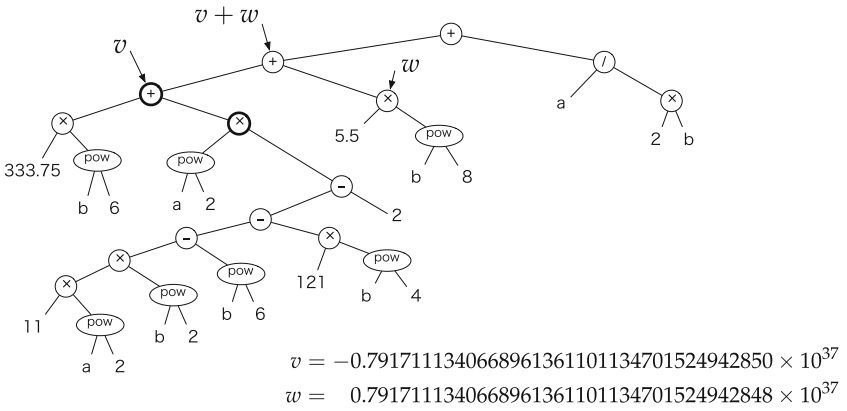

Fig. 1. Tree representation of Rump's example. Catastrophic cancellation happens at node labeled "$v + w$" if an insufficient number of bits are used for computation.

Multiple-precision floating-point arithmetic such as that defined in IEEE 754 [2] has been used to avoid catastrophic cancellation, and some processors have been designed to support multiple-precision arithmetic. In addition, arbitrary-precision arithmetic operations can be performed by using libraries such as the GNU MPFR library [3].

However, even if multiple-precision floating-point numbers are used, the accuracy of the computational results cannot be guaranteed. A simple example of this is *Rump's example* [4] (Fig. 1):

$$y = 333.75b^6 + a^2(11a^2b^2 - b^6 - 121b^4 - 2) + 5.5b^8 + a/(2b)\ . \tag{1}$$

If $a = 77617$ and $b = 33096$, $a^2 = 5.5b^2 + 1$ holds. Thus, (1) is transformed to

$$y = -2 + a/(2b) = -54767/66192 \doteqdot -0.827396059946821368 14.$$

However, calculating this equation using double-precision numbers yields $y = -1.1805916207174113 \times 10^{21}$, which is far from the correct answer. This inaccuracy is due to catastrophic cancellation of significant digits. In fact, a completely useless value is produced by the addition of two numbers with absolute values that are almost the same (the leading 36 digits in their decimal are identical) and the signs are opposite at the node labeled "$v + w$" in Fig. 1.

The result of quadruple-precision computation of (1) using the MPFR library is $y = 1.172603940053179$, and the result with higher precision computation with 122 bits for the mantissa of each number is $y = -0.8273960599468214$, which is sufficiently accurate. However, a closer look at the computational process reveals that a 122-bit length for *all* operations is too much — only two operations (indicated by thick circles in Fig. 1) are critical enough to require a 122-bit mantissa. For other operations, quadruple-precision numbers suffice.

Next, let us slightly change the value of b. The results of (1) with $a = 77617$ and $b = 33095$ are $y = -4.7833916866560586 \times 10^{32}$ for double-precision and

$y = -4.783391686660554 \times 10^{32}$ for a 122-bit length mantissa — there is no substantial difference between them.

This small example illustrates the inherent difficulties of floating-point computation with a fixed number of digits.

- It is difficult to determine a suitable number of bits for each operation.
- It is difficult to detect cancellation of significant digits from the obtained result.
- It is difficult to pinpoint the operations in a program that caused cancellation.
- It is difficult to predict the effect of changes in the data.

To obtain results that satisfy the required accuracy, it is necessary to properly estimate the mantissa length for each floating-point operation. Unfortunately, it is quite difficult to statically decide the optimal precision for each operation. Thus, we take a *dynamic* approach; the partial results in floating-point representations are incrementally improved using adaptive control. This *adaptive accuracy refinement* process requires tracing of accumulated errors and application of suitable strategies to remove inaccurate values.

In this paper, we present a mechanism for adaptive accuracy refinement of computational results. It uses lazy lists, each one containing an infinite sequence of specially designed floating-point numbers with gradually improving accuracies. These numbers are approximations of the same real value and are ordered on the basis of what we call "accuracy." In computation using this mechanism, "referring to the next value" corresponds to obtaining a better (more accurate) value via recomputation of subexpressions, which are propagated to dependent operands automatically. In principle, computation using this *improving floating-point number* (IFN) mechanism is applicable to any numerical algorithm.

To get the flavor how our IFN mechanism works, let us return to Rump's example. Here, we use our implementation of the mechanism in Haskell, which will be described in Sect. 3. First, we define the function *rump* in Haskell.

```
rump :: Fractional t ⇒ t → t → t
rump a b = 333.75 * b^6 + a^2 * (11 * a^2 * b^2 − b^6 − 121 * b^4 − 2)
           + 5.5 * b^8 + a / (2 * b)
```

The following is a GHCi session that calls *rump* for *Double* numbers (incorrect value is returned) and then calls *rump* for IFNs.

```
*Main> rump 77617 33096
−1.1805916207174113e21
*Main> let qs = rump (77617 :: IFN) (33096 :: IFN)
*Main> accurateValue 32 qs
−0.827396059946821 x 10 ^ (0) : ac=55
*Main> accurateValue 128 qs
−0.827396059946821368141165095479816292 00 x 10 ^ (0) : ac=130
```

Once we bind the result of *rump* that uses IFN computation to the variable *qs*, we can obtain an arbitrary accuracy of the resultant value by giving the desired accuracy to the library function *accurateValue*.

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

- Our proposed IFN mechanism achieves adaptive accuracy refinement for the computation of each subexpression in the entire expression. This enables the user to easily obtain a computational result with the desired accuracy without cancellation of significant digits. In contrast to computation using a fixed number of bits such as with functions in the MPFR library, computation using the IFN mechanism can be done using only a sufficient number of bits for each subexpression. Note that "sufficient number of bits" varies with the subexpression; the number of bits for each subexpression is automatically adjusted.
- We formalize an IFN as a list of specially represented floating-point numbers that approximate the same real value and for which the accuracies are improved. On the basis of this concept, we define unary / binary operators and basic mathematical functions on IFNs, the results of which are also IFNs.
- An experimental IFN library was implemented as both a Haskell library and a pure C library. The lazy evaluation facility of Haskell facilitated implementation of the Haskell library in a quite natural manner because computation using IFNs proceeds in response to demands for "more accurate" values of the IFN of interest. A C version of the library was developed to cope with the performance problem of the Haskell library. Application of these libraries to numerical problems for which it is difficult to obtain precise answers by using *Double* numbers demonstrated the effectiveness of the proposed IFN mechanism.

The organization of this paper is as follows. Section 2 outlines the concept of our IFN mechanism and the adaptive refinement of subexpressions using IFNs. In Sect. 3, we describe the details of IFNs, floating-point numbers for IFNs, and arithmetic operations for IFNs. In Sect. 4, support for logical expressions is discussed. In Sect. 5, issues related to implementing IFN libraries are considered. Section 6 describes several numerical examples demonstrating the effectiveness and applicability of our IFN mechanism. In Sect. 7, we discuss a few IFN-related issues. Related work is covered in Sect. 8. Finally, we conclude with a brief summary in Sect. 9.

Throughout this paper, we use Haskell [5] with some extra typesetting features to describe the design of IFN libraries. Though some datatypes in Sects. 3 and 4 are defined naively for the sake of conciseness, our practical IFN Haskell library uses more efficient implementation by means of Haskell's foreign function interface (FFI), as described in Sect. 5.

2 Improving Floating-Point Numbers

To improve the accuracy of floating-point computations of an expression by adaptive and appropriate accuracy refinements of its subexpressions, we introduce the concept of *improving floating-point numbers*. Intuitively speaking, an IFN is an

infinite sequence of floating-point values, each of which approximates the same real value v, i.e., the ideal result of the computation. The further to the right the value is in the sequence, the closer it is to v.

The basic idea of IFNs came from the notion of "improving sequences" [6,7], which is a finite monotonic sequence of approximation values of a final value that are gradually improved in accordance with an ordering relation. However, IFNs are different from improving sequences because IFNs are *infinite* sequences.

Hereafter, we denote a floating-point number by q. Each q is associated with an integer that represents its accuracy, which we denote by $ac\ q$. We represent an IFN by using a lazy list of q's.

$$qs = [q_0, q_1, q_2, \ldots]$$

Since accuracies are improved in an IFN, the relation $ac\ q_i < ac\ q_{i+1}$ holds for all $i \geq 0$. To see why an IFN has to be infinite, consider the case in which $v = 5/7$. Since $5/7 = (0.101101...)_2$, instances of IFNs that represent v could be

$$[(0.1)_2, (0.11)_2, (0.110)_2, (0.1011)_2, (0.10111)_2, (0.101110)_2, \ldots]$$
$$[(0.110)_2, (0.101110)_2, (0.101101110)_2, \ldots].$$

In any case, an infinite sequence is capable of expressing arbitrary accuracies naturally.

The floating-point values in an IFN are forced from left to right. If all q_i's $(i \leq k)$ have been forced and q_j's $(k < j)$ have not been forced yet, the *current value* of this IFN is q_k, and its current accuracy is $ac\ q_k$. If the current accuracy is unsatisfactory, q_{k+1} is forced, and the current value is set to q_{k+1}. This process is repeated until the desired accuracy is obtained.

To see how the computation in terms of IFNs proceeds, let us consider a simple example: addition of v and w. Suppose that v corresponds to $ps = [p_0, p_1, \ldots, p_h, \ldots]$ where its current value is p_h, and w corresponds to $qs = [q_0, q_1, \ldots, q_k, \ldots]$ where its current value is q_k. Also suppose that $v + w$ corresponds to $rs = [r_0, r_1, \ldots, r_l, \ldots]$ where we have already computed $r_0, r_1, \ldots, r_l$ by using $p_0, p_1, \ldots, p_h, q_0, q_1, \ldots, q_k$. If we are not satisfied with $ac\ r_l$, we want to compute r_{l+1} to obtain better accuracy. If the next values of ps and qs (p_{h+1} and q_{k+1}) are judged to be necessary, we force them and compute r_{l+1} by using both p_{h+1} and q_{k+1}. It is worth noting that the computation of r_{l+1} has to produce a value with better accuracy than r_l. If ps is the result of another computation, say multiplication of ps' and ps'', forcing p_{h+1} induces other floating-point values in ps' and ps'' to be forced. In this way, the entire computation is driven by the propagation of the demands for the next values in IFNs. This kind of demand-driven computation can be naturally described on the basis of lazy evaluation. We thus used Haskell to build a prototype IFN library. As we describe in later sections, there are several design choices in a practical implementation.

3 Adaptive Accuracy Refinement with IFNs

3.1 Floating-Point Datatype

Let us define datatype Q for floating-point numbers:

$$\textbf{data}\ Q = Q\ \{\ sign :: Int,\ \ mantissa :: [\,Int\,],\ \ expo :: Int\ \}$$

where *sign*, *mantissa*, and *expo* are the sign, mantissa, and exponent, respectively; *sign* is either 1 or -1, and *mantissa* is a finite list for which the elements are either 0 or 1. We assume that q is an instance of Q.

Functions *ac* and *lenM* are defined as follows.

$$\begin{array}{lcl} ac,\ lenM :: & Q \to Int \\ ac\ q & = & lenM\ q - expo\ q \\ lenM\ q & = & length\ (mantissa\ q) \end{array}$$

Each q is associated with two real values, $\langle q \rangle$ and $\{q\}$:

$$\langle q \rangle = (sign\ q) \times \left(\frac{b_1}{2^1} + \frac{b_2}{2^2} + \cdots + \frac{b_n}{2^n} \right) \times 2^e, \quad \{q\} = \frac{1}{2^{n+1}} \times 2^e,$$

where $e = expo\ q$, $[b_1, b_2, \ldots, b_n] = mantissa\ q$, and $n = lenM\ q$. Using $\langle q \rangle$ and $\{q\}$, we define the relation between a real value v and q. If v satisfies the following inequality, we say that a floating-point number q is an *approximation* of v and write $v \leftharpoonup q$. We also say that v is *properly represented* by q if $v \leftharpoonup q$ holds.

$$\langle q \rangle - \{q\} \ \leq v \leq\ \langle q \rangle + \{q\}$$

Note that for each q, the range within which a real value being properly represented by q should reside is rigorously defined only by q. Here, the following property holds:

$$\{q\} = \frac{1}{2^{(ac\ q)+1}}.$$

This means that the accuracy of q, $ac\ q$, indicates the width of the range in which real values approximated by q can reside. A larger $ac\ q$ indicates that q represents a real v satisfying $v \leftharpoonup q$ more precisely.

Example 1. Consider the following three floating-point numbers q, q', and q''. For q and q', equations $\langle q \rangle = \langle q' \rangle$ and $ac\ q = ac\ q'$ hold. In addition, for any real v, $v \leftharpoonup q \iff v \leftharpoonup q'$. However, for q and q'', $\langle q \rangle = \langle q'' \rangle$, but $ac\ q < ac\ q''$. For any real v, $v \leftharpoonup q'' \implies v \leftharpoonup q$ but $v \leftharpoonup q \not\Longrightarrow v \leftharpoonup q''$.

$$\begin{array}{l} q\ \ = Q\ \{\ sign = 1,\ mantissa = [1,1,0,0],\ expo = -2\} \\ q'\ = Q\ \{\ sign = 1,\ mantissa = [0,0,0,1,1,0,0],\ expo = 1\} \\ q'' = Q\ \{\ sign = 1,\ mantissa = [1,1,0,0,0,0,0,0,0],\ expo = -2\} \end{array}$$

Here, let us define *left-shifting* and *right-shifting*. Suppose q_1, q_2, and q_3 are instances of Q:

$$q_1 = Q\ \{\ sign = s,\ mantissa = [b_1, b_2, \ldots, b_n],\ expo = e\}$$
$$q_2 = Q\ \{\ sign = s,\ mantissa = [b_{m+1}, b_{m+2}, \ldots, b_n],\ expo = e - m\}$$
$$q_3 = Q\ \{\ sign = s,\ mantissa = [\underbrace{0, 0, \ldots, 0}_{m}, b_1, b_2, \ldots, b_n],\ expo = e + m\}$$

Left-shifting q_1 by m digits leads to q_2, where $lenM\ q_2 = n - m$, and right-shifting q_2 by m digits leads to q_3, where $lenM\ q_3 = n + m$. Note that right-shifting does not change real values associated with Q, i.e., $\langle q_1 \rangle = \langle q_3 \rangle$ and $\{q_1\} = \{q_3\}$. If $b_1 = b_2 = \cdots = b_m = 0$ and $b_{m+1} \neq 0$, left-shifting q_1 by m digits removes all leading zeros of the mantissa of q_1. We call the removal of leading zeros of the mantissa by left-shifting *normalization*. Normalization also does not change real values associated with Q. In Example 1, q and q' have the same representation under normalization.

3.2 Definition of IFNs

An IFN is an infinite list of Q's instances $qs = [q_0, q_1, q_2, \ldots]$ that satisfies two conditions:

- $ac\ q_i < ac\ q_{i+1}$ holds for all $i \geq 0$, and
- there is a real value v that satisfies $v \leftharpoonup q_i$ for all $i \geq 0$.[1]

If every element of an IFN qs properly represents a real v, i.e., $v \leftharpoonup q_i$ for all $i \geq 0$, we say that *qs is an IFN with respect to v* and write $v \leftharpoonup qs$ by overloading the relation symbol $\leftharpoonup$. Since IFNs are infinite lists, $v \leftharpoonup qs \wedge w \leftharpoonup qs \Longrightarrow v = w$.

Now let us define datatype *IFN* for IFNs in Haskell.

type $IFN = [Q]$

An IFN instance can be generated from a literal in a program by using

$$genIFNfromString\ s\ =\ map\ (\backslash n \rightarrow fromString\ n\ s)\ [initN,\ initN + diffN\ ..]$$

The function $fromString :: Int \rightarrow String \rightarrow Q$ generates an instance of Q that approximates a given number in a string representation with a designated length the mantissa. *initN* and *diffN* are respectively the initial length of the mantissa and the difference between the lengths of the mantissas of consecutive elements in the returned IFN. As will be described in Sect. 5.2 in detail, the values of *initN* and *diffN* and the way in which the IFNs are generated greatly affect the performance of the IFN library.

Given an IFN, we can obtain an instance of Q for which the accuracy is equal to or greater than a specified value by using the *accurateValue* function.

$$accurateValue\ a\ qs = head\ (dropWhile\ \ (\backslash q \rightarrow ac\ q < a)\ qs)$$

[1] The condition is weaker than the following: $\langle q_i \rangle - \{q_i\} \leq \langle q_{i+1} \rangle - \{q_{i+1}\} < \langle q_{i+1} \rangle + \{q_{i+1}\} \leq \langle q_i \rangle + \{q_i\}$; i.e., each successive floating-point number does not necessarily denote a sub-interval of the previous one.

We call functions that accept IFNs as operands and returns an IFN *IFN operators*. When a program is constructed by composition of IFN operators, results at any level of accuracy can be obtained by using *accurateValue*.

3.3 Floating-Point Arithmetic Operators for Q

Arithmetic operators for Q are expected to satisfy several conditions. That is, for any q_1 and q_2 and real v_1 and v_2,

$$\begin{aligned} v_1 \leftarrowtail q_1 &\Longrightarrow -v_1 \leftarrowtail \mathit{negQ}\ q_1 \\ v_1 \leftarrowtail q_1 \wedge v_2 \leftarrowtail q_2 &\Longrightarrow v_1 + v_2 \leftarrowtail \mathit{addQ}\ q_1\ q_2 \\ v_1 \leftarrowtail q_1 \wedge v_2 \leftarrowtail q_2 &\Longrightarrow v_1 \times v_2 \leftarrowtail \mathit{mulQ}\ q_1\ q_2 \\ v_1 \leftarrowtail q_1 \wedge v_2 \leftarrowtail q_2 &\Longrightarrow v_1/v_2 \leftarrowtail \mathit{divQ}\ q_1\ q_2, \end{aligned}$$

where *negQ*, *addQ*, *mulQ*, and *divQ* correspond to real operators $-$ (unary minus), $+$, $\times$, and $/$, respectively.

We define *negQ* as follows. It is easy to check that the above condition for *negQ* holds since $\langle \mathit{negQ}\ q_1 \rangle = -\langle q_1 \rangle$ and $\{\mathit{negQ}\ q_1\} = \{q_1\}$.

$$\begin{aligned} &\mathit{negQ}\ \ :: Q \to Q \\ &\mathit{negQ}\ q = Q\ \{\ \mathit{sign} = -\ (\mathit{sign}\ q),\ \mathit{mantissa} = \mathit{mantissa}\ q,\ \mathit{expo} = \mathit{expo}\ q\ \} \end{aligned}$$

We have to be careful in defining binary operators for Q because simple definitions would not satisfy the above conditions.

Example 2. Let us consider addition operators for Q. We can readily define a function for variable-precision addition, *simpleAddQ*, such that $\mathit{simpleAddQ}\ q_1\ q_2$ is an instance of Q satisfying $\langle \mathit{simpleAddQ}\ q_1\ q_2 \rangle = \langle q_1 \rangle + \langle q_2 \rangle$. Now, for the following instances of Q, $v_1 \leftarrowtail q_1$ and $v_2 \leftarrowtail q_2$ hold where $v_1 = (0.7813)_{10}$ and $v_2 = (0.3907)_{10}$. However, $v_1 + v_2 = (1.1720)_{10} \not\leftarrowtail q_{1+2} = \mathit{simpleAddQ}\ q_1\ q_2$.

$$\begin{aligned} q_1 &= Q\ \{\ \mathit{sign} = 1,\ \mathit{mantissa} = [1,1,0,1],\ \mathit{expo} = 0\} \\ q_2 &= Q\ \{\ \mathit{sign} = 1,\ \mathit{mantissa} = [1,1,0,1],\ \mathit{expo} = -1\} \\ q_{1+2} &= Q\ \{\ \mathit{sign} = 1,\ \mathit{mantissa} = [1,0,0,1,1,1],\ \mathit{expo} = 1\} \end{aligned}$$

In contrast, "rounding" of q_{1+2} with a mantissa length of 3 yields q'_{1+2}, which satisfies the relation $v_1 + v_2 \leftarrowtail q'_{1+2}$.

$$q'_{1+2} = Q\ \{\ \mathit{sign} = 1,\ \mathit{mantissa} = [1,0,1],\ \mathit{expo} = 1\}$$

Rounding of Q is defined as follows. Let q_a, q_b, and q_c be instances of Q:

$$\begin{aligned} q_a &= Q\ \{\ \mathit{sign} = s,\ \mathit{mantissa} = [b_1, b_2, \ldots, b_r, b_{r+1}, \ldots, b_n],\ \mathit{expo} = e\} \\ q_b &= Q\ \{\ \mathit{sign} = s,\ \mathit{mantissa} = [b_1, b_2, \ldots, b_r],\ \mathit{expo} = e\} \\ q_c &= Q\ \{\ \mathit{sign} = s,\ \mathit{mantissa} = [\underbrace{0, 0, \ldots, 0}_{r-1}, b_{r+1}],\ \mathit{expo} = e\}. \end{aligned}$$

Then, rounding of q_a at r-th digit is computed as $\mathit{simpleAddQ}\ q_b\ q_c$.

As illustrated in the above example, "proper" floating-point operators apparently do not generate results with a very long mantissa. Let us denote the rounded representation of q at the r-th digit as $roundQ\ q\ r$. In Example 2, $q'_{1+2} = roundQ\ q_{1+2}\ 3$.

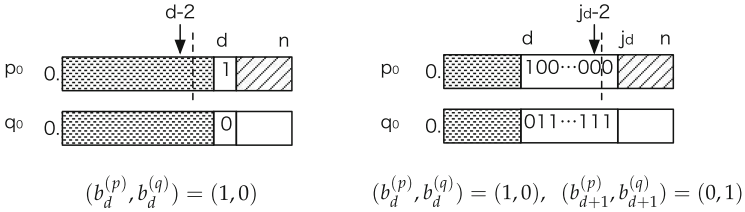

Fig. 2. Position of mantissa to be rounded to construct $r_0 = p_0 \sqcup_0 q_0$. Bits in shaded rectangles of p_0 and q_0 are the same. Other bits that are not explicitly shown are arbitrary. If $(b_{d+1}^{(p)}, b_{d+1}^{(q)}) = (0,1)$, j_d is defined as in the right figure.

Now we show the definition of *addQ* in terms of *simpleAddQ* used in Example 2. First we briefly describe a binary operator $\sqcup$ of type $Q \to Q \to Q$, which is defined using another binary operator $\sqcup_0$ of type $Q \to Q \to Q$. Suppose that two Qs, namely p_0 and q_0, are as follows:

$$p_0 = Q\ \{\ sign = s_p,\ mantissa = [b_1^{(p)}, b_2^{(p)}, \ldots, b_n^{(p)}],\ expo = e\}$$
$$q_0 = Q\ \{\ sign = s_q,\ mantissa = [b_1^{(q)}, b_2^{(q)}, \ldots, b_n^{(q)}],\ expo = e\}.$$

Then, $r_0 = p_0 \sqcup_0 q_0$ is an instance of Q such that among all qs that satisfy both $\langle p_0 \rangle \leftharpoonup q$ and $\langle q_0 \rangle \leftharpoonup q$, r_0 gives the maximum of $ac\ q$ for most cases. The function $\sqcup_0$ is defined as follows.

- Case $\langle p_0 \rangle > \langle q_0 \rangle \geq 0$ or $\langle p_0 \rangle < \langle q_0 \rangle \leq 0$: Suppose the following expressions hold, as shown in Fig. 2:

$$\begin{cases} b_i^{(p)} = b_i^{(q)}, & i = 1, \ldots, d-1 \\ b_i^{(p)} \neq b_i^{(q)}, & i = d. \end{cases}$$

  Note that $(b_d^{(p)}, b_d^{(q)}) = (1,0)$. Here, we suppose $d \geq 3$ assuming that p_0 and q_0 are appropriately shifted beforehand. Now, if $d < n$ and $(b_{d+1}^{(p)}, b_{d+1}^{(q)}) = (0,1)$, then $r_0 = roundQ\ p_0\ (j_d - 2)$, where j_d is the smallest integer satisfying $j_d > d+1$ and $(b_{j_d}^{(p)}, b_{j_d}^{(q)}) \neq (0,1)$. If $d = n$ or $(b_{d+1}^{(p)}, b_{d+1}^{(q)}) \neq (0,1)$, then $r_0 = roundQ\ p_0\ (d-2)$.
- Case $\langle q_0 \rangle > \langle p_0 \rangle \geq 0$or $\langle q_0 \rangle < \langle p_0 \rangle \leq 0$: Exchange p_0 and q_0 and apply the previous case.
- Case $|\langle p_0 \rangle| = |\langle q_0 \rangle| > 0$: If $\langle p_0 \rangle = \langle q_0 \rangle$, $r_0 = p_0 = q_0$. If $\langle p_0 \rangle = -\langle q_0 \rangle$, $r_0 = roundQ\ p_0\ (d-2)$, where the first $d-1$ digits of $mantissa\ p_0$ are zeros and the d-th digit is 1.

– Case $|\langle p_0 \rangle| = |\langle q_0 \rangle| = 0$: $r_0 = p_0 = q_0$.

Now, the binary operator $\sqcup$ is defined using $\sqcup_0$ and appropriate shifting to support an arbitrary exponent: if $q_3 = q_1 \sqcup q_2$, q_3 is an instance of Q such that, among all qs that satisfy both $\langle q_1 \rangle \leftarrow\!\!\!\!\!\!\cdot\;\; q$ and $\langle q_2 \rangle \leftarrow\!\!\!\!\!\!\cdot\;\; q$, q_3 gives the maximum of $ac\ q$ for most cases. Then, $addQ\ q_1\ q_2$ is defined as $(simpleAddQ\ h\ d) \sqcup (simpleAddQ\ h\ (negQ\ d))$, where $h = simpleAddQ\ q_1\ q_2$, $d = simpleAddQ\ d_1\ d_2$, $\langle d_1 \rangle = \{q_1\}$, and $\langle d_2 \rangle = \{q_2\}$. The above condition for $addQ$ holds because the following equations hold.

$$\begin{aligned} \langle simpleAddQ\ h\ d \rangle &= \langle h \rangle + \langle d \rangle = (\langle q_1 \rangle + \{q_1\}) + (\langle q_2 \rangle + \{q_2\}) \\ \langle simpleAddQ\ h\ (negQ\ d) \rangle &= \langle h \rangle - \langle d \rangle = (\langle q_1 \rangle - \{q_1\}) + (\langle q_2 \rangle - \{q_2\}) \end{aligned}$$

Note that the d_1 and d_2 instances of Q are easily constructed and that the entire computation of $addQ$ can be done without evaluating real values $\langle q_1 \rangle$, $\langle q_2 \rangle$, $\{q_1\}$, and $\{q_2\}$.

Likewise, $mulQ$ can be defined. First, we define $simpleMulQ$, where $simpleMulQ\ q_1\ q_2$ is an instance of Q such that $\langle simpleMulQ\ q_1\ q_2 \rangle = \langle q_1 \rangle \times \langle q_2 \rangle$. Then, for q_1 and q_2 such that $\langle q_1 \rangle > 0$ and $\langle q_2 \rangle > 0$, $mulQ\ q_1\ q_2$ is defined as

$$\begin{aligned} &(simpleAddQ\ (simpleAddQ\ h\ d_3)\ (simpleAddQ\ d_1\ d_2)) \\ &\quad \sqcup (simpleAddQ\ (simpleAddQ\ h\ (negQ\ d_1))\ (simpleAddQ\ (negQ\ d_2)\ d_3), \end{aligned}$$

where $h = simpleMulQ\ q_1\ q_2$, $d_1 = simpleMulQ\ q_1\ d_1'$, $d_2 = simpleMulQ\ q_1\ d_2'$, $\langle d_1' \rangle = \{q_2\}$, $\langle d_2' \rangle = \{q_1\}$, and $\langle d_3 \rangle = \{q_1\} \times \{q_2\}$. The definitions of $mulQ$ for other cases are similar.

We can also define $divQ$ in a similar manner; its definition is omitted due to space limitation.

As described above, binary Q operators such as $addQ$ and $mulQ$ are constructed using raw arithmetic operators such as $simpleAddQ$ and $simpleMulQ$ and a binary operator $\sqcup$ for accuracy management. The raw operators are expected to be built with highly tuned variable-precision libraries such as MPFR.

3.4 Unary IFN Operator: Negation

Negation of an IFN is constructed as follows:

$$\begin{aligned} &negIFN :: IFN \to IFN \\ &negIFN = map\ negQ \end{aligned}$$

If qs is an IFN with respect to a real v, $negIFN\ qs$ should be an IFN with respect to $-v$. First, the ordering of accuracy can be preserved if

$$ac\ q < ac\ q' \Longrightarrow ac\ (negQ\ q) < ac\ (negQ\ q')$$

holds. The $negQ$ defined in Sect. 3.3 satisfies this condition because $ac\ (negQ\ p) = ac\ p$. In addition, $\langle negQ\ p \rangle = -\langle p \rangle$. Thus, $negIFN$ is indeed a proper IFN operator for negation.

3.5 Binary IFN Operators

First, we define addition of two IFNs.

$$
\begin{array}{lll}
addIFN & :: IFN \to IFN \to IFN & \\
addIFN & = addIFN'\ a_{\text{init}} \textbf{ where } a_{\text{init}} = \text{minimum value of } Int & \\
addIFN' & :: Int \to IFN \to IFN \to IFN & \\
addIFN'\ a\ (p{:}ps)\ (q{:}qs) & & \\
\quad \mid ac\ p < ac\ q = & \textbf{if } a' \leq a \textbf{ then } addIFN'\ a\ ps\ (q{:}qs) & \\
& \textbf{else } r : addIFN'\ a'ps\ (q{:}qs) & \\
\quad \mid ac\ p > ac\ q = & \textbf{if } a' \leq a \textbf{ then } addIFN'\ a\ (p{:}ps)\ qs & \\
& \textbf{else } r : addIFN'\ a'(p{:}ps)\ qs & \\
\quad \mid \textbf{otherwise} = & \textbf{if } a' \leq a \textbf{ then } addIFN'\ a\ ps\ qs & \\
& \textbf{else } r : addIFN'\ a'ps\ qs & \\
\quad \textbf{where } r = & addQ\ p\ q & \\
a' = & ac\ r &
\end{array}
$$

The function *addIFN′* keeps the current accuracy (the accuracy of the current value) in its first argument and uses it to ensure that the accuracy of the next value in the resultant IFN is improved.[2]

One may wonder whether recursive application of *addIFN'* while obtaining "the next value" might not terminate. However, *addIFN* is shown to work properly as follows.

Let us examine the behavior of *addIFN*. Suppose we are adding two IFNs, $ps = [p_0, p_1, \ldots]$ and $qs = [q_0, q_1, \ldots]$, and the result is $[r_0, r_1, \ldots] = addIFN\ ps\ qs$, which has a current value of $r_i = addQ\ p_j\ q_k$. If r_{i+1} is forced, *addIFN′* behaves as follows:

- If $ac\ p_j < ac\ q_k$, the candidate for r_{i+1} is $addQ\ p_{j+1}\ q_k$. In this case, the inequality $min\ (ac\ p_j)\ (ac\ q_k) < min\ (ac\ p_{j+1})\ (ac\ q_k)$ holds. If $ac\ r_i < ac\ (addQ\ p_{j+1}\ q_k)$, then r_{i+1} is readily available as $addQ\ p_{j+1}\ q_k$. On the other hand, if $ac\ r_i \geq ac\ (addQ\ p_{j+1}\ q_k)$, searching for appropriate operands continues.
- If $ac\ p_j > ac\ q_k$, the candidate for r_{i+1} is $addQ\ p_j\ q_{k+1}$. Here, $min\ (ac\ p_j)\ (ac\ q_k) < min\ (ac\ p_j)\ (ac\ q_{k+1})$ holds. If $ac\ r_i \geq ac\ (addQ\ p_j\ q_{k+1})$, searching for appropriate operands continues.
- If $ac\ p_j = ac\ q_k$, the candidate for r_{i+1} is $addQ\ p_{j+1}\ q_{k+1}$. In this case, $min\ (ac\ p_j)\ (ac\ q_k) < min\ (ac\ p_{j+1})\ (ac\ q_{k+1})$ holds. If $ac\ r_i \geq ac\ (addQ\ p_{j+1}\ q_{k+1})$, searching for appropriate operands continues.

Here we show that the search terminates and that r_{i+1} is eventually available. *addQ* has the following property; its derivation is omitted for the sake of brevity:

$$min\ (ac\ p)\ (ac\ q) - 3 \leq ac\ (addQ\ p\ q) \leq min\ (ac\ p)\ (ac\ q) - 1,$$

[2] The inequality $a' \leq a$ in the definition of *addIFN'* can be modified to enhance performance; see Sect. 5.2 for details.

where p and q are normalized instances of Q. From this inequality, we can see that $addQ\ p\ q$ depends on the value of $min\ (ac\ p)\ (ac\ q)$. The following property is also derived from the above inequality:

$$min\ (ac\ p)\ (ac\ q) < min\ (ac\ p')\ (ac\ q') - 2 \Longrightarrow ac\ (addQ\ p\ q) < ac\ (addQ\ p'\ q').$$

As described above, calculating the next value by using *addIFN'* is done on the basis of the ascending order of the value of $min\ (ac\ p)\ (ac\ q)$ without an upper limit. Thus, the accuracy of $addQ\ p\ q$ can be greater than any designated integer. As a result, the above property guarantees the termination of *addIFN'*. In summary, *addIFN* is a proper IFN operator for addition.

Next, we define multiplication of two IFNs.

$$\begin{array}{lll}
mulIFN & :: IFN \to IFN \to IFN & \\
mulIFN & = mulIFN'\ a_{\text{init}} \textbf{ where } a_{\text{init}} = \text{minimum value of } Int & \\
mulIFN' & :: Int \to IFN \to IFN \to IFN & \\
mulIFN'\ a\ (p{:}ps)\ (q{:}qs) & & \\
\quad \mid lenM\ p < lenM\ q = \textbf{if } a' \leq a \textbf{ then } mulIFN'\ a\ ps\ (q{:}qs) & & \\
\qquad\qquad\qquad\qquad\qquad\quad \textbf{else } r : mulIFN'\ a'ps\ (q{:}qs) & & \\
\quad \mid lenM\ p > lenM\ q = \textbf{if } a' \leq a \textbf{ then } mulIFN'\ a\ (p{:}ps)\ qs & & \\
\qquad\qquad\qquad\qquad\qquad\quad \textbf{else } r : mulIFN'\ a'(p{:}ps)\ qs & & \\
\quad \mid \textbf{otherwise} = \textbf{if } a' \leq a \textbf{ then } mulIFN'\ a\ ps\ qs & & \\
\qquad\qquad\qquad\qquad\qquad\quad \textbf{else } r : mulIFN'\ a'ps\ qs & & \\
\qquad\qquad \textbf{where } r = mulQ\ p\ q & & \\
\qquad\qquad\qquad\quad a' = ac\ r & &
\end{array}$$

To show that *mulIFN* works properly, first we show that *mulIFN'* searches for appropriate operands for use in calculating "the next value" in such a way that the value of $f\ p\ q = ac\ p + ac\ q - max\ (lenM\ p)\ (lenM\ q)$ increases, where p and q are instances of Q that are used to calculate $mulQ\ p\ q$.

Suppose we are multiplying two IFNs, $ps = [p_0, p_1, \ldots]$ and $qs = [q_0, q_1, \ldots]$, and the result is $[r_0, r_1, \ldots] = mulIFN\ ps\ qs$, which has a current value of $r_i = mulQ\ p_j\ q_k$. The behavior of *mulIFN'* when r_{i+1} is forced is as follows:

- If $lenM\ p_j < lenM\ q_k$, the candidate for r_{i+1} is $mulQ\ p_{j+1}\ q_k$. Let us examine the value of $f\ p_{j+1}\ q_k\ - f\ p_j\ q_k$. If $lenM\ p_{j+1} > lenM\ q_k$, then

$$\begin{aligned} f\ p_{j+1}\ q_k - f\ p_j\ q_k &= ac\ p_{j+1} - ac\ p_j - lenM\ p_{j+1} + lenM\ q_k \\ &= (expo\ p_j - expo\ p_{j+1}) + (lenM\ q_k - lenM\ p_j) > 0 \end{aligned}$$

  since $expo\ p_j \geq expo\ p_{j+1}$. On the other hand, if $lenM\ p_{j+1} \leq lenM\ q_k$, then

$$f\ p_{j+1}\ q_k - f\ p_j\ q_k = ac\ p_{j+1} - ac\ p_j - lenM\ q_k + lenM\ q_k > 0.$$

  If $ac\ r_i < ac\ (mulQ\ p_{j+1}\ q_k)$, then r_{i+1} is readily available as $mulQ\ p_{j+1}\ q_k$. If $ac\ r_i \geq ac\ (mulQ\ p_{j+1}\ q_k)$, searching for appropriate operands continues.
- If $lenM\ p_j > lenM\ q_k$, $f\ p_j\ q_{k+1} > f\ p_j\ q_k$ can be shown similarly. If $ac\ r_i \geq ac\ (mulQ\ p_j\ q_{k+1})$, searching for appropriate operands continues.

– If $lenM\ p_j = lenM\ q_k$, $f\ p_{j+1}\ q_{k+1} > f\ p_j\ q_k$ can be shown similarly. If $ac\ r_i \geq ac\ (mulQ\ p_{j+1}\ q_{k+1})$, searching for appropriate operands continues.

Now, letting p, p', q, and q' be normalized instances of Q, the property of *mulQ*, the derivation of which is omitted for the sake of brevity, is summarized as

$$f\ p\ q < f\ p'\ q' - 2 \Longrightarrow ac\ (mulQ\ p\ q) < ac\ (mulQ\ p'\ q').$$

The above property guarantees that $ac\ (mulQ\ p\ q)$ can be greater than any designated integer. Thus, as described above, the termination of *mulIFN'* is guaranteed. In summary, *mulIFN* is a proper IFN operator for multiplication.

subIFN is defined by a composition of *addIFN* and *negIFN*.

$$\begin{aligned} &subIFN && :: IFN \to IFN \to IFN \\ &subIFN\ ps\ qs = && addIFN\ ps\ (negIFN\ qs) \end{aligned}$$

Division operator *divIFN* can be essentially constructed in the same way.

3.6 Basic Mathematical Functions for IFN

In this section, we briefly sketch the implementation of basic mathematical functions for IFN such as exponential (*expIFN*), square root (*sqrtIFN*), logarithmic (*logIFN*), and trigonometric (*sinIFN*, *cosIFN*, etc.) functions. Each IFN function is constructed using corresponding arithmetic operators for Q, e.g., *expQ* for *expIFN* and *sqrtQ* for *sqrtIFN*. As described in Sect. 3.3, these mathematical functions should satisfy the following conditions:

$$\begin{aligned} v \leftarrow q &\Longrightarrow e^v \leftarrow expQ\ q \\ v \leftarrow q \wedge v \geq 0 &\Longrightarrow \sqrt{v} \leftarrow sqrtQ\ q \\ v \leftarrow q \wedge v > 0 &\Longrightarrow \log\ v \leftarrow logQ\ q \\ v \leftarrow q &\Longrightarrow \sin\ v \leftarrow sinQ\ q, \end{aligned}$$

where v is a real value and q is an instance of Q. To implement the Q functions, we make use of the MPFR functions. Note that the result of each MPFR function is guaranteed to be the nearest possible floating-point value assuming that its inputs are exact values [3].

Monotonic (and continuous) functions including *expQ*, *sqrtQ*, and *logQ* can be easily implemented. Since the exact upper and lower bounds for which q (an instance of Q) properly represents are readily obtained from q, the projected range can be computed using a unary MPFR function, and the corresponding Q value can be generated using the operator $\sqcup$. For example, *expQ* can be defines as:

$$\begin{aligned} &expQ && :: && Q \to Q \\ &expQ && q = && l \sqcup u \\ &\textbf{where}\ && l = && simpleExpQ\ (simpleAddQ\ q\ (negQ\ d)) \\ &&& u = && simpleExpQ\ (simpleAddQ\ q\ d) \\ &&& d = && Q\ \{\ sign = 1,\ mantissa = [1],\ expo = -\ ac\ q\ \} \end{aligned}$$

where *simpleExpQ* corresponds to the square root function of MPFR. Note that $\{q\} = \langle d \rangle$.

The functions described here are unary. The definitions of monotonic IFN functions are straightforward; e.g., $expIFN = map\ expQ$.

Non-monotonic functions such as trigonometric functions are not as easy to implement as monotonic ones. The implementation of continuous functions such as *sinQ* and *cosQ* requires special treatment for the maximum and / or minimum values in the range of each Q. If the accuracy of an input value is too low, the corresponding range could be $(-1, 1)$. Other trigonometric functions such as *tanQ* can be composed of other functions. Although such special treatment has to be taken into consideration, non-monotonic functions can be implemented by using MPFR functions (although we have not yet implemented them in the current IFN library.)

To construct non-monotonic IFN functions, the resultant lazy list must be filtered such that its elements are in ascending order of accuracy. For example, *sinIFN* can be defined as follows:

$$
\begin{array}{lcl}
sinIFN & :: & IFN \to IFN \\
sinIFN & = & sinIFN'\ a_{\text{init}}\ \textbf{where}\ a_{\text{init}} = -1 \\
sinIFN' & :: & Int \to IFN \to IFN \\
sinIFN'\ a\ (p{:}ps) & = & \textbf{if}\ a' \leq a\ \textbf{then}\ sinIFN'\ a\ ps\ \textbf{else}\ r : sinIFN'\ a'\ ps \\
\quad \textbf{where}\ r & = & sinQ\ p \\
a' & = & ac\ r
\end{array}
$$

3.7 Precision and Accuracy

In ordinary floating-point computations, the word "precision" usually denotes the number of (meaningful) digits in the mantissa. Precision thus implies the relative error of each floating-point number. In contrast, we use "accuracy" for the index of preciseness of each floating-point number of Q. As defined in Sect. 3.1, the accuracy of a floating-point number q of type Q, $ac\ q$, is an indicator of the absolute error of q. In the strict sense, precision and accuracy are different. However, phrases used in the context of numerical computation such as "precision improvement" and "accuracy improvement" imply the same meaning. Hereafter, we use the "precision" and "accuracy" interchangeably unless otherwise stated.

4 Logical Expressions with IFNs

4.1 Treatment of Logical Expressions

The preciseness of the results of numerical computations does not depend only on the accuracy of the arithmetic operations. For some numerical methods including iterative solving and truncated summation of an infinite series, control transfers are required to terminate the computation. Thus, correct evaluation of branch conditions, i.e., logical expressions, dependent on the results of floating-point arithmetic is crucial.

Logical expressions have type *Bool*. Since there is no "more precise *True*," or "*True* but very close to *False*," a logical expression should return a simple Boolean value. Boolean values should not change even when adaptive accuracy refinement is applied to arithmetic expressions. This means that accurate comparisons of floating-point values have to be done immediately.

Zero-testing for a floating-point value is not as easy as one might think. It might not be enough to simply look at the mantissa of floating-point numbers. Recall that every floating-point number is simply an approximation of a real value. In fact, algorithms dependent on the result of equality checking are considered to be improper when writing numerical programs. Nevertheless, two numbers are frequently compared.

Taking the above into account, our design decision is summarized as follows:

- True zero is separately treated throughout the computation.
- Zero-testing is done in accordance with an auxiliary parameter.

4.2 Introduction of True Zeros

Here we introduce the idea of *true zero*s. A true zero is generated when a literal constant zero appears in a program. In addition, a true zero can be the result of an operation on Q. For example, we can extend $mulQ$ so that $mulQ\ q\ z = z$, where z is a true zero. A true zero is never generated from an operation when all operands are not true zeros.

To treat true zeros, we extend the datatype Q:

$$\textbf{data}\ Q = Q\ \{\ sign :: Int,\ \ mantissa :: [\,Int\,],\ \ expo :: Int,\ \ zeroFlag :: Bool\ \}$$

We let Q include *zeroFlag*, which represents whether the value is a true zero. Notice that we do not care about the values of *sign*, *mantissa*, and *expo* for a true zero because any operation on Q can be defined without them.

As for IFN operators, if we define $ac\ z = \infty$ and $lenM\ z = \infty$ for a true zero z, no modification for the IFN operators defined in Sects. 3.4 and 3.5 is required. In fact, when a true zero appears as the current value in an IFN, the next value need not be accessed because there should be no better value.

As a result of this separate handling of true zeros, a true zero can appear only as the first element in a lazy list, and the subsequent elements are never accessed. This property can be used for optimization in the implementation of IFN operators. For example, $addIFN\ \ ps\ \ qs$ can be defined to return ps immediately if the first element of qs is a true zero.

4.3 Zero-Testing and Equality Testing

Besides true zero, it is practically convenient to capture very small computational results as "approximated zero." Here, *tol* is an integer called *tolerance* that is used to judge whether to treat a value as zero.

If we assume that the q instance of Q is normalized, a zero-testing function can be defined as:

$$
\begin{array}{llll}
\mathit{zeroQ} :: Q \to \mathit{Int} \to \mathit{Int} \\
\mathit{zeroQ}\ q\ \mathit{tol} & \mid \mathit{zeroFlag}\ q \quad \| \quad \mathit{expo}\ q < \mathit{tol} & = 1 \\
& \mid \mathit{mantissa}\ q \neq [\,]\ \&\&\ \mathit{expo}\ q \geq \mathit{tol} & = -1 \\
& \mid \textbf{otherwise} & = 0
\end{array}
$$

$\mathit{zeroQ}\ q\ \mathit{tol} = 0$ indicates that $\langle q \rangle = 0$ but $\mathit{expo}\, q$ is too large to treat q as zero.

By using zeroQ, a zero-testing function for IFNs with tolerance tol can be defined as:

$$
\begin{array}{lll}
\mathit{zeroIFN} :: \mathit{IFN} \to \mathit{Int} \to \mathit{Bool} \\
\mathit{zeroIFN}\ (q{:}qs)\ \ \mathit{tol} & \mid \mathit{zeroQ}\ q\ \mathit{tol} == 1 & = \mathit{True} \\
& \mid \mathit{zeroQ}\ q\ \mathit{tol} == -1 & = \mathit{False} \\
& \mid \textbf{otherwise} & = \mathit{zeroIFN}\ qs\ \mathit{tol}
\end{array}
$$

$\mathit{zeroIFN}$ uses adaptive accuracy refinement on a given IFN. $\mathit{zeroIFN}$ stops for any IFN. Letting $v \leftarrowtail qs$, the relationship between an IFN qs and an integer tol is

$$
\begin{cases}
\mathit{zeroIFN}\ qs\ \mathit{tol} = \mathit{True} & \Longrightarrow |v| < 2^{tol-1} \\
\mathit{zeroIFN}\ qs\ \mathit{tol} = \mathit{False} & \Longrightarrow |v| \geq 2^{tol-1}.
\end{cases}
$$

Equality testing of two IFNs is defined as follows.

$$
\begin{array}{ll}
\mathit{equalIFN} & :: \mathit{IFN} \to \mathit{IFN} \to \mathit{Int} \to \mathit{Bool} \\
\mathit{equalIFN}\ ps\ qs\ \mathit{tol} & = \ \mathit{zeroIFN}\ (\mathit{subIFN}\ ps\ qs)\ \mathit{tol}
\end{array}
$$

where subIFN is an IFN operator defined in Sect. 3.5.

The relationship among ps, qs, and tol is

$$
\begin{cases}
\mathit{equalIFN}\ ps\ qs\ \mathit{tol} = \mathit{True} & \Longrightarrow |v - w| < 2^{tol-1} \\
\mathit{equalIFN}\ ps\ qs\ \mathit{tol} = \mathit{False} & \Longrightarrow |v - w| \geq 2^{tol-1},
\end{cases}
$$

where $v \leftarrowtail ps$ and $w \leftarrowtail qs$. Note that $\mathit{equalIFN}$ does not test true equality; it tests only the approximate equality of two IFNs.

4.4 Comparison Between Two IFNs

A comparison function that judges whether an IFN is less than another is defined as:

$$
\begin{array}{ll}
\mathit{lessIFN} & :: \mathit{IFN} \to \mathit{IFN} \to \mathit{Int} \to \mathit{Bool} \\
\mathit{lessIFN}\ \ (p{:}ps)\ \ (q{:}qs)\ \ \mathit{tol} \\
\quad \mid \langle p \rangle + \{p\} < \langle q \rangle - \{q\} & = \mathit{True} \\
\quad \mid \langle p \rangle - \{p\} \geq \langle q \rangle + \{q\} & = \mathit{False} \\
\quad \mid \mathit{zeroQ}\ (\mathit{addQ}\ q\ (\mathit{negQ}\ p))\ \mathit{tol} == 1 & = \mathit{False} \\
\quad \mid \mathit{ac}\ p > \mathit{ac}\ q & = \mathit{lessIFN}\ (p{:}ps)\ \ qs\ \ \mathit{tol} \\
\quad \mid \mathit{ac}\ p < \mathit{ac}\ q & = \mathit{lessIFN}\ ps\ (q{:}qs)\ \ \mathit{tol} \\
\quad \mid \textbf{otherwise} & = \mathit{lessIFN}\ ps\ qs\ \ \mathit{tol}
\end{array}
$$

lessIFN uses adaptive accuracy refinement on a given IFN, and stops for any IFN as well as *zeroIFN*. The relationship between ps, qs and tol is

$$\begin{cases} lessIFN\ ps\ qs\ tol = True \Longrightarrow v + 2^{tol-1} \leq w \\ lessIFN\ ps\ qs\ tol = False \Longrightarrow v + 2^{tol-1} > w, \end{cases}$$

where $v \leftharpoonup ps$ and $w \leftharpoonup qs$.

Other comparison operators can be defined using *lessIFN*:

$$\begin{array}{ll} lesseqIFN,\ greaterIFN,\ greatereqIFN & :: IFN \to IFN \to Int \to Bool \\ lesseqIFN & :: IFN \to IFN \to Int \to Bool \\ lesseqIFN\ \ (p{:}ps)\ \ (q{:}qs)\ \ tol & \\ \quad \mid \langle p \rangle + \{p\} < \langle q \rangle - \{q\} & = True \\ \quad \mid \langle p \rangle - \{p\} \geq \langle q \rangle + \{q\} & = False \\ \quad \mid zeroQ\ (addQ\ q\ (negQ\ p))\ tol == 1 & = True \\ \quad \mid ac\ p > ac\ q & = lesseqIFN\ (p{:}ps)\ \ qs\ \ tol \\ \quad \mid ac\ p < ac\ q & = lesseqIFN\ ps\ (q{:}qs)\ \ tol \\ \quad \mid \textbf{otherwise} & = lesseqIFN\ ps\ qs\ \ tol \\ greaterIFN\ ps\ qs\ \ tol & = not\ (lesseqIFN\ ps\ qs\ \ tol) \\ greatereqIFN\ ps\ qs\ \ tol & = not\ (lessIFN\ ps\ qs\ \ tol) \end{array}$$

5 Detailed Design and Implementation of Primitives

In Sects. 3 and 4, we presented the principle of IFN and the basic design of primitives in the IFN library. In fact, the functions in Haskell presented in previous sections could literally constitute a set of library primitives. However, the data structures and algorithms used in their codes are not suitable for high-speed computation. In this section, we discuss implementation issues and the detailed design of primitives.

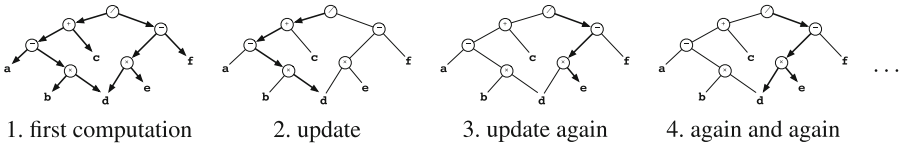

Fig. 3. Adaptive precision refinement in naive way. Arrows along edges depict propagation of order for recomputation.

5.1 Adaptive Accuracy Refinement for IFN Computations

Control of Adaptive Accuracy Refinement. In previous sections, we described the basic design of IFNs. An IFN does not know whether its current value is sufficiently accurate; this can be judged only at the root of the computation tree. If the root judges that the accuracy of the IFN's current value at the root is unsatisfactory, it issues a demand for "recomputation for a more accurate

value" to its child (or children) on the basis of the definition of the operation (e.g., *addIFN*) at the root. This recomputation demand is propagated down to the (part of the) leaves. Every node that receives the demand produces the next more accurate value in the IFN stream it returns. The entire computation proceeds as a repetition of this *accuracy refinement process*.

The accuracy refinement process, illustrated in Fig. 3, starts only if the resulting precision does not meet the user's requirement. The demand for recomputation is propagated all the way to one or more leaves, and the values at the nodes on the paths from the leaves to the root are updated. This process is iterated until the resultant value at the root is sufficiently accurate. Even though recomputation is done only at nodes of a subtree in each iteration, the repetition process is inherently inefficient.

In fact, the accuracy refinement process could be performed too many times because each node in the computation tree is not informed of the required accuracies of the value it produces (in the IFN stream). Our approach to reducing this inefficiency is to advise each node of the required accuracy of the value it produces when propagating the demand for a more accurate value. Although the required accuracy at the root is not actually required at other nodes, we use the accuracy given by the user as the lower limit at each node of the computation tree. Although this is a simple heuristic approach, it works well for many cases from our experience.

Setting Initial Accuracy of IFNs. To control the initial precisions of the IFNs at the nodes and leaves of the computation tree, we introduce datatype *IFNgen* for constructing the computation tree.

```
type IFNgen = Int → IFN
```

IFNgen is a function type that accepts an accuracy and generates an IFN for which the initial element's accuracy is defined by the argument. Numerical programs are constructed using "*IFNgen* operators" instead of IFN operators. This modification is realizable by replacing several primitives as follows:

```
addIFN          :: IFNgen → IFNgen → IFNgen
addIFN v w a = addIFN' a (v a) (w a)

genIFNfromString      :: String → IFNgen
genIFNfromString s a = map f [a, a+ diffN ..]
            where f n = fromString (fromIntegral n) s

accurateValue      :: Int → IFNgen → Q
accurateValue a v = head $ dropWhile(\q → ac q < a) (v a)
```

Although the types of primitives change, the whole course of computation is consistently replaced, and the numerical programs prepared by the user do not need to be modified.

With *IFNgen*, the result accuracy required by the user is propagated all the way through the edges of the computation tree to the leaves. With this

functionality, the modification of the accuracy required for operands at each IFN operator can be controlled in detail by using an appropriate function, *nextAccuracy*, that returns, given the current accuracy, the next accuracy to be attained. For example, *addIFN* can be defined as follows:

```
addIFN          :: IFNgen → IFNgen → IFNgen
addIFN v w a = let a'= nextAccuracy a in addIFN' a (v a') (w a')
```

Conversion of an *IFN*-based library to an *IFNgen*-based library causes subtle but non-ignorable problems. When computation trees are constructed on the basis of *IFNgen*, the nodes of the tree are type *IFNgen* functions that generate specific IFNs at run-time. Therefore, even though the generator of type *IFNgen* is referred to by multiple *IFNgen* operators as the generator of the operands, the IFNs generated at run-time are not shared. The use of non-shared IFNs means that complicated numerical programs that utilize iterative algorithms and / or matrix computations are unfeasible because of the blowup of duplicated computations.

To share IFNs among IFNgen operators, we use a static storage area to save once-generated IFNs. This mechanism is achieved by modifying several primitives as follows. Each *IFNgen* operator creates a reference *IORef Maybe IFN* when it is first evaluated, and the created IFNs are written into it. Note that the user's program does not need not to be modified.

```
import Data.IORef
saveIFN    :: IFNgen → IFNgen
saveIFN f =  unsafePerformIO $ do
    ref ← newIORef Nothing
    let  f' a = unsafePerformIO $ do
              t ← readIORef ref
              case  t  of
                Nothing → do let v = f a
                             writeIORef ref (Just v)
                             return v
                Just v' → do let newv = dropWhile (\n → ac n < a) v'
                             return newv
    return  f'

addIFN      ::  IFNgen → IFNgen → IFNgen
addIFN v w =  saveIFN $ \a → let a'= f a in addIFN' a (v a') (w a')

genIFNfromString    :: String → IFNgen
genIFNfromString s =  saveIFN $ \a → map f [a, a + diffN..]
          where f n =  fromString (fromIntegral n) s
```

5.2 Configuration of IFNs

The definition of IFN is very simple as described in Sect. 3.2. To optimize performance, we can design IFN operators as follows:

- The accuracy of each IFN's initial element for literal values can be set higher than the user's required accuracy for the total results.
- The difference in accuracy between the elements of each IFN for literal values can be increased, or even the sequence can be defined on the basis of a geometric series.
- A lower limit on accuracy refinement can be imposed for each IFN operator.

In our experience, these modifications and parameter settings greatly affect the performance of numerical programs. However, it is difficult to determine the optimal configuration. If the accuracy of each IFN's initial element for literals is too large, useless computations to obtain too accurate values by variable-length floating-point operations might take a very long time. However, if it is too small, there could be a huge number of iterations for adaptive accuracy refinement, resulting in a great amount of time to obtain accurate results. The settings for other items in the list above also affect computational cost. Deciding the appropriate configuration remains for future work.

5.3 Structure of Datatype Q and Design of Q Operators

The datatype definition of Q presented in Sect. 3.1 was introduced for the sake of concise explanation of the idea of IFNs and is not sufficient for efficient implementation of IFNs. It is quite natural to define Q in terms of variable-length floating point representation in C. Thus, we decided to use the MPFR library [3] for this purpose. We used the MPFR data structure *__mpfr_struct* to construct Q in C and used MPFR routines like *mpfr_add* to define Q operators such as *simpleAddQ*. The definition of Q in C is as follows.

```
typedef struct {
  __mpfr_struct *_mp;
  bool _zeroFlag;
} Q;
```

We used Boehm's GC library for memory management of C objects. By using Haskell's foreign function interface (FFI) and the *Foreign* and *Foreign.Concurrent* modules, we made Haskell's GC cooperate with Boehm's GC.

5.4 Implementation in C Without General-Purpose Garbage Collectors

The adaptive accuracy refinement functionality of the IFN library relies greatly on the lazy evaluation of infinite lists. However, because the dynamic structure constructed at run-time is fairly simple, general-purpose garbage collectors are not necessary (or may be unsuitable). Objects created at run-time do not cyclically reference to each other, and the point of destruction of an IFN's elements in the course of execution is predeterminable. Thus, an implementation of the IFN library in C with its own memory management, such as collection by reference

counting, may outperform the awkward cooperation of Haskell's and Boehm's GC facilities described in Sect. 5.4, especially for complicated problems.

On the basis of this perspective, we implemented a version of the IFN library completely in C without general-purpose garbage collectors. IFN cells and other objects such as instances of Q and their mantissas are managed by using a set of ring buffers.

6 Numerical Experiments

Examples presented in this section demonstrate the potential and applicability of adaptive accuracy refinement using IFNs. The results are compared with those of programs using other computable real arithmetic libraries, namely Exact Real Arithmetic (ERA) (version 1.0) [8] and iRRAM (version 2013_01) [9].

ERA is a Haskell library for computable real numbers developed by David Lester [8]. In the library, a computable real, say x, is represented by a function f of type $Int \to Integer$, where the following inequality holds for all $i :: Int, i \geq 0$.

$$\frac{f\ i-1}{2^i} < x < \frac{f\ i+1}{2^i}$$

Once f is constructed, the user can obtain an approximated value of x that possesses the desired accuracy. Numerical operators are built on type *CReal*. For example, addition is essentially defined as follows.

$$\begin{array}{l}
\textbf{data}\ \mathit{CReal} = \mathit{CR}\ (\mathit{Int} \to \mathit{Integer}) \\
(+) \qquad\qquad\qquad :: \mathit{CReal} \to \mathit{CReal} \to \mathit{CReal} \\
(\mathit{CR}\ x') + (\mathit{CR}\ y') = \mathit{CR}\ (\backslash p \to \mathit{round_uk}\ ((x'\ (p+2) + y'(p+2))\ \%\ 4)) \\
\textbf{where}\ \mathit{round_uk}\ x = \mathit{floor}\ (x + 1\ \%\ 2)
\end{array}$$

Although the implementation is done in quite a small number of lines, ERA is thought to be the fastest among computable real libraries implemented in Haskell [10]. ERA is not based on the idea of adaptive refinement of accuracy — it does not need it. However, to obtain a final result with user-defined accuracy, much computation on integers may be required.

iRRAM is a C++ library for computable real numbers [9]. Each variable of type *REAL* is constructed with a multiple-precision number and information on its absolute error. Errors are accumulated during the course of computation, and if the error in the resultant value exceeds the user's request, the entire computation is repeated with a significantly better precision iRRAM uses external multiple-precision libraries such as MPFR. It was the "clear winner" of a competition among several systems for exact arithmetic held at CCA 2000 [11].

Note that IFN libraries are still prototypes and have much room for optimization. Although the comparison among the libraries reported here was done mainly on the basis of performance, the aim was not to determine a "winner" but to clarify the characteristics of the libraries.

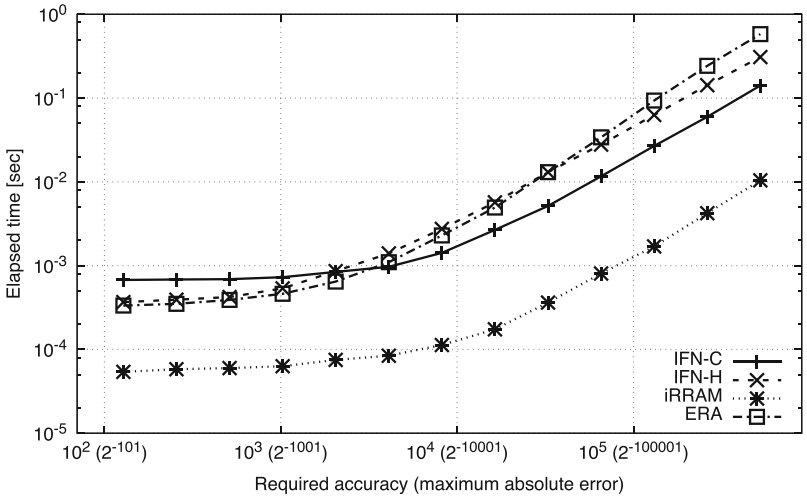

Fig. 4. Performance of libraries for Rump's example. For IFN-C and IFN-H, required accuracies were from 128 to 524,288 (i.e., required absolute errors were from 2^{-129} to $2^{-524289}$). For ERA and iRRAM, required number of decimal digits for result ranged from 38 to 157,826.

6.1 Environment

All programs were run on a MacBook Pro (Intel Core i7 3 GHz with 16GB memory) running OS X 10.10.3. We used two versions of the IFN library. One was implemented in Haskell (IFN-H), described in Sects. 5.1 and 5.3, and the other was implemented in C (IFN-C), described in Sect. 5.4. Both used MPFR [3].

Application programs were written in Haskell for IFN-H and ERA, in C for IFN-C, and in C++ for iRRAM. The Haskell programs given to IFN-H and ERA were the same, and the C and C++ programs were equivalent to the Haskell programs.

The software versions were GHC 7.8.4, Boehm GC 7.4.2 (for IFN-H), GCC 4.8.4, MPFR 3.1.1, and GMP 5.1.1.

6.2 Example 1: Simple Expression

Rump's example (1) was evaluated for accuracies from 128 to 524,288, where required absolute errors were from 2^{-129} to $2^{-524289}$. For ERA and iRRAM, required accuracy was translated into required number of decimal digits for the result, and it ranged from 38 to 157,826. The program used for Haskell was as follows, where type T was *IFNgen* for IFN-H and *CReal* for ERA.

```
rump (77617::T) (33096::T)
where
  rump a b = ...
```

The elapsed time to perform one evaluation of the program against the required accuracy is plotted in Fig. 4. All the curves depict essentially the same trend, indicating that the performance of IFN is comparable to those of the others.

IFN-C outperformed IFN-H and ERA for a higher range of required accuracy. However, when used for obtaining moderately accurate results with maximum absolute errors of, say, less than 2^{-1024}, IFN-H performed better than IFN-C. Since the behaviors of the numerical computations of IFN-C and IFN-H were the same, it should be possible to raise the performance of IFN-C to the level of IFN-H.

6.3 Example 2: Solving a Sensitive Problem

The Hilbert matrix is a square matrix $[h_{ij}]$, 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 requiring several accuracies. We used LU factorization algorithm without pivoting. The C and C++ programs for IFN-C and iRRAM were written as follows. The Haskell programs for IFN-H and ERA were written using mutable arrays offered by *Data.Array.IO*.

$$
\begin{array}{l}
\mathit{for}\ \ k = 1,\ \ \ldots,\ \ n \\
\quad \mathit{for}\ \ i = k + 1,\ \ldots,\ \ n \\
\quad\quad a_{ik} \leftarrow a_{ik}/a_{kk} \\
\quad\quad \mathit{for}\ \ j = k + 1,\ \ldots,\ \ n \\
\quad\quad\quad a_{ij} \leftarrow a_{ij} - a_{ik}a_{kj}
\end{array}
$$

As shown in Fig. 5, the behavior of IFN-C was similar to that of iRRAM. Although clear differences are apparent, IFN-C can be said to be comparable to iRRAM. IFN-H performed as well as IFN-C. However, there was substantial performance degradation of IFN-H for large problems. This is probably because the cost of garbage collection increased with the problem size.

The poor performance of ERA might be because the computed values of common subexpressions were not shared in the overall computation. Although the same functions were used at nodes in the computation tree, the fact that multiple invocations of these functions did not share the results caused an explosion in the amount of computation. A computational real arithmetic library of this type is thus difficult to use in algorithms that perform iterations or matrix computations.

Effects of the Accuracy Control Strategy. Here, we show the results for Hilbert systems with 64 unknowns for several configuration patterns (cf. Sect. 5.2). The experiments were done using IFN-C. The choices used were as follows:

1. The accuracy of the first element's in each IFN for literal values was set as either

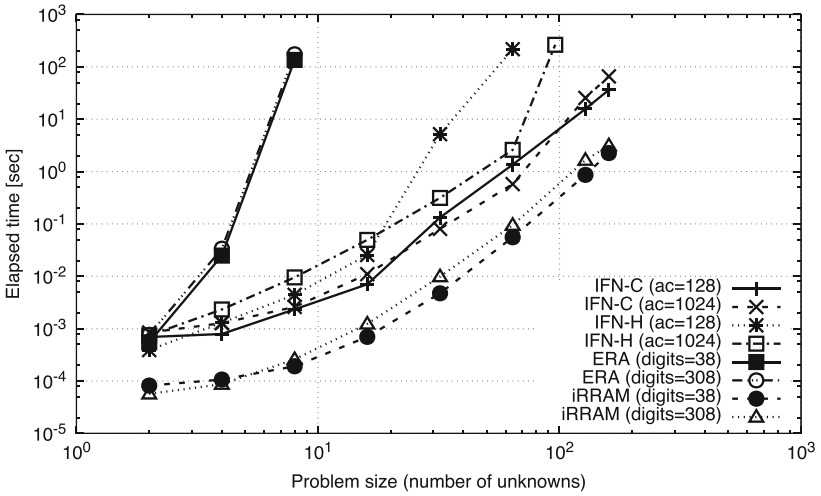

Fig. 5. Solving linear equations with Hilbert matrix. Required accuracies were 128 and 1024 (i.e., required maximum absolute errors were 2^{-129} and 2^{-1025}). For ERA and iRRAM, required correct figures in decimal were 38 and 308.

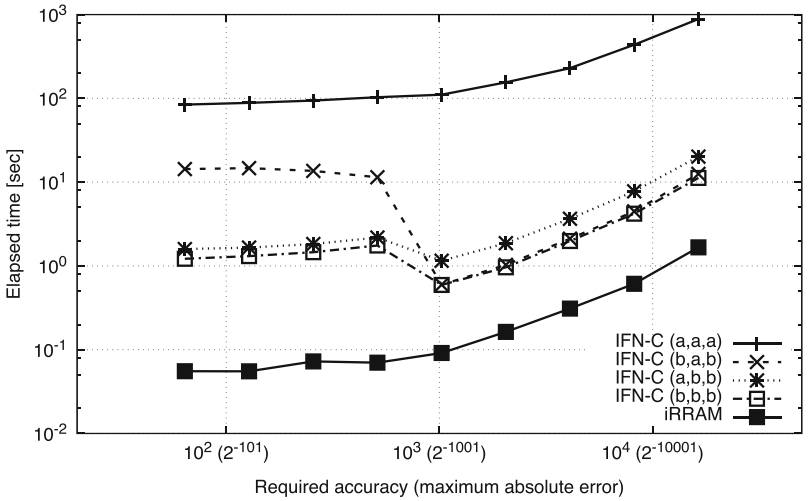

Fig. 6. Solving linear equations with Hilbert matrix of size 64×64. Required accuracies for IFN-C were from 64 to 16,384 (i.e., required absolute errors were from 2^{-65} to 2^{-16385}). For iRRAM, required number of decimal digits for the result ranged from 20 to 4,932. IFN-C performance was evaluated for four configurations.

(a) the accuracy required for the result or
(b) twice the required accuracy.

2. The pattern to increase the accuracy of each IFN for literal values was set as either

(a) an arithmetic sequence with a common difference of 64 or
(b) a geometric series with a common ratio of 2.

3. The minimum increase in accuracy imposed between consecutive elements in all IFNs was set as either (a) 4 or (b) 32.

We describe each configuration using triples; for example, (a, b, a) denotes that the choices for items 1, 2, and 3 are (a), (b), and (a), respectively. The results for IFN-C and IFN-H shown in Figs. 4 and 5 were obtained with a configuration pattern of (b, b, b).

The results with patterns (a, a, a), (b, a, b), (a, b, b), and (b, b, b) are shown in Fig. 6. The results for iRRAM are also shown for reference. The performance with (b, b, b) was about two orders of magnitude better than that with (a, a, a). The curves for (b, a, b) and (b, b, b) were almost the same when the required accuracy was greater than 1,024 (required maximum absolute error less than 2^{-1025}). In fact, all executions with (b, _, _) that were configured with large initial accuracy values for literals did not perform any recomputation in the range of requirements, including (b, b, a) and (b, a, b), which are not shown in Fig. 6. When moderate accuracy was required, items 2 and 3 both have to be (b). Although not shown in Fig. 6, the result with (b, b, a) was very close to that of (b, a, b).

The performance with configuration pattern (b, b, b) was the best. This seems to be mainly due to the reduced number of invocations for recomputation. However, the most suitable configuration may greatly depend on the application. For example, there could be cases where reducing the number of computations increases total elapsed time because of the cost required for too accurate variable-precision floating-point operations. Detailed analysis of the behavior of IFN libraries for other applications is left for future work.

7 Discussion

7.1 Usability

In principle, adaptive accuracy refinement using IFNs is applicable to almost all numerical computations. One of the major tasks for implementing an IFN version of a numerical program is basically to replace floating-point operators in the original program with their corresponding IFN operators (or IFNgen operators). As for the Haskell library we developed, since the *IFN* datatype is declared as instances of number-related type classes such as *Num* and *Fractional*, the user can use normal arithmetic operators such as $+$, $*$ without any knowledge of the internal details of IFNs.

7.2 Applicability

IFNs can be used to solve any type of numerical problem. As demonstrated in Sect. 6, evaluation of complex expressions and matrix computations can be carried out.

Usage of IFNs simply enables the user to eliminate errors caused by the usage of fixed length floating-point representations. When IFNs are used for programs derived from an approximated modeling process such as truncation, discretization, and any style of simplification, the accuracy of the results is not guaranteed. In addition, because the accuracy of zero-testing depends on the external parameter, the accuracy of the entire result can also depend on the parameter. This includes iterative algorithms such as the Newton-Raphson method, which require comparison of (computed) real values to control the execution.

7.3 Performance

Compared to arithmetic on the basis of a fixed-number of bits such as *Double*, IFN arithmetic has inherent and significant overhead caused by operators on Q (e.g., *addQ*), operators on IFNs (e.g., *addIFN*) and the control of demand driven computation.

The properties of IFN operators enable the user to obtain computational results for expressions composed of basic arithmetic operations with the desired accuracies. However, expressions that would cause catastrophic cancellations of significant digits (if *Double* numbers were used) could take a long computational time with IFNs.

The results of our experimental implementation of IFN libraries (Sect. 6) show the potential of IFNs. Although current versions of IFN libraries cannot be used to solve large problems, we think there is much room for improvement in terms of computational speed. Rearrangement of the many bit-wise operations required for post-processing of $\sqcup$ in each Q operator may be one way to reduce computational time.

8 Related Work

The basic idea of IFNs came from the notion of *improving sequences* [6,7]. An improving sequence is a finite monotonic sequence of approximation values of a final value that are improved gradually in accordance with an ordering relation. Programs constructed with improving sequences offers many opportunities to eliminate too accurate computation. The effectiveness of improving sequences has been demonstrated for combinatorial optimization problems. IFNs are different from improving sequences because IFNs are specialized and *infinite* streams representing real numbers

Dynamic detection of catastrophic cancellation of significant digits can be realized by monitoring each normalization process of the resultant values in floating-point arithmetic (an example of a vector processor with cancellation detectors is presented elsewhere [12]).

Sophisticated tools for detecting precision degradation have been proposed [13,14]. They use *shadow values* calculated using higher precision arithmetic, so the results are presumably better. By using the tools, the occurrence of cancellation can be detected and causes of the errors can be analyzed. However, accuracy of the results will never be guaranteed by those tools.

An option other than floating-point arithmetic for carrying out precise numerical computation is classical rational arithmetic. Each value is represented by a pair of integers, i.e., a numerator and a denominator. While it can be used to evaluate simple expressions like Rump's example (Sect. 1), it may not be suitable for complicated programs due to huge computational cost. Approximation by a kind of rounding may be needed.

Interval arithmetic, in which each value is represented by upper and lower bounds, is another tool for accuracy-aware numerical computation [1]. Analyses of complicated functions and linear systems based on special facilities for precise inner product computation have been carried out [15]. Our floating-point representation of Q can be considered a virtual interval representation. The feasibility of adaptive refinement with interval arithmetic based on lazy evaluation has been indirectly confirmed by our research.

Exact real computer arithmetic has been studied for decades [16]. To deal with reals, a number of representations have been considered, such as continued fraction representation [17,18], linear fractional transformations for exact arithmetic [19], radix representations with negative digits, non-integral or irrational bases, and nested sequences of rational intervals [20]. In scaled-integer representation [21], reals are represented by functions, and each arithmetic operation consists of the application and construction of functions with rational computation for coefficients. Some of those adopt lazy stream implementation in which each infinite stream represents a real. Although we have not examined in them detail, several libraries have been implemented [8,9,22–24]. Compared to previous approaches, ours is simple and thus has many design choices for the details. The most significant difference between IFNs and others is that each approximate value is represented as a precision-guaranteed floating-point number. In that sense, an IFN can be seen as a sequence of intervals. It can also be seen as a kind of Cauchy sequence although different from ordinary definitions of Cauchy sequences for computable real arithmetic.

9 Conclusion

We have demonstrated a means of adaptive refinement based on lazy evaluation for accurate floating-point numerical computations. We presented the idea of improving floating-point numbers (IFNs) to encapsulate adaptive refinement processes into lazy lists. Computations using IFNs was described in detail. Numerical results based on implementations in Haskell and C demonstrated the effectiveness of the approach.

Future work includes optimization, design of supporting tools, and numerical evaluation on a larger scale of complicated numerical problems.

Acknowledgments. The first author thanks Toshiaki Kitamura and Mizuki Yokoyama for their useful discussions. The authors thank the anonymous reviewers for their detailed comments. This work was supported in part by an HCU Grant for Special Academic Research (General Studies) under Grant No.1030301.

References

1. Knuth, D.E.: The Art of Computer Programming, Sect. 4.2.2, 3rd edn. Addison-Wesley, Boston (1997)
2. Microprocessor Standards Committee of the IEEE Computer Society: IEEE Standard for Floating-Point Arithmetic. IEEE Standard 754 (2008)
3. The GNU MPFR Library. http://www.mpfr.org
4. Rump, S.M.: Algorithms for verified inclusion. In: Moore, R. (ed.) Reliability in Computing, Perspectives in Computing, pp. 109–126. Academic Press, New York (1988)
5. Bird, R.: Introduction to Functional Programming Using Haskell, 2nd edn. Prentice Hall, Englewood Cliffs (1998)
6. Morimoto, T., Takano, Y., Iwasaki, H.: Instantly turning a naive exhaustive search into three efficient searches with pruning. In: Hanus, M. (ed.) PADL 2007. LNCS, vol. 4354, pp. 65–79. Springer, Heidelberg (2007)
7. Iwasaki, H., Morimoto, T., Takano, Y.: Pruning with improving sequences in lazy functional programs. Higher-Order Symbolic Comput. **24**, 281–309 (2011)
8. Lester, D.: ERA: Exact Real Arithmetic, version 1.0 (2000). http://hackage.haskell.org/package/numbers-3000.2.0.1/docs/Data-Number-CReal.html
9. Müller, N.T.: The iRRAM: exact arithmetic in C++. In: Blank, J., Brattka, V., Hertling, P. (eds.) CCA 2000. LNCS, vol. 2064, p. 222. Springer, Heidelberg (2001)
10. Haskell wiki, Applications and Libraries / Mathematics, Sect. 3.2.2.2. http://www.haskell.org/haskellwiki/Applications_and_libraries/Mathematics#Dynamic_precision_by_lazy_evaluation
11. Blanck, J.: Exact real arithmetic systems: results of competition. In: Blank, J., Brattka, V., Hertling, P. (eds.) CCA 2000. LNCS, vol. 2064, p. 389. Springer, Heidelberg (2001)
12. Aniya, S., Kitamura, T.: A performance improvement for floating-point arithmetic unit with precision degradation detection. In: The 17th Workshop on Synthesis And System Integration of Mixed Information Technologies (SASIMI 2012), pp. 490–491 (2012)
13. Jeffrey, K.H., Lam, M.O., Stewart, G.W.: Dynamic floating-point cancellation detection. In: WHIST 2011 (2011)
14. Benz, F., Hildebrandt, A., Hack, S.: A dynamic program analysis to find floating-point accuracy problems. In: SIGPLAN Notices, PLDI 2012, vol. 47, No. 6, pp. 453–462 (2012)
15. Jansen, P., Weidner, P.: High-accuracy arithmetic software – some tests of the ACRITH problem-solving routines. ACM Trans. Math. Softw. **12**(1), 62–70 (1986)
16. Gowland, P., Lester, D.R.: A survey of exact arithmetic implementations. In: Blank, J., Brattka, V., Hertling, P. (eds.) CCA 2000. LNCS, vol. 2064, pp. 30–47. Springer, Heidelberg (2001)
17. Gosper, W.: Continued fractions (1972). http://www.inwap.com/pdp10/hbaker/hakmem/cf.html
18. Vuillemin, J.: Exact real computer arithmetic with continued fractions. IEEE Trans. Comput. **39**, 1087–1105 (1990)
19. Potts, P.: Exact real arithmetic using Möbius transformations, Ph.D. thesis, Department of Computing, Imperial College of Science, Technology and Medicine, University of London. http://www.doc.ic.ac.uk/%7Eae/papers.html
20. Escardó, M.: Introduction to exact numerical computation, Notes for a tutorial at ISSAC (2000). http://www.cs.bham.ac.uk/%7Emhe/issac

21. Boehm, H.-J., Cartwright, R., Riggle, M., O'Donnell, M.J.: Exact real arithmetic: a case study in higher order programming. In: ACM Symposium on Lisp and Functional Programming, pp. 162–173 (1986)
22. Guy, M.: bignum / BigFloat (2007). http://bignum.sourceforge.net/
23. Edalat, A.: Exact real number computation using linear fractional transformations, Final Report on EPSRC grant GR/L43077/01 (2001)
24. Lambov, B.: RealLib: an efficient implementation of exact real arithmetic. Math. Struct. Comput. Sci. **17**, 81–98 (2007)