Electronic Preprint for Journal of Information Processing Vol.29

Regular Paper

Rumpfr: A Fast and Memory Leak-free Rust Binding to the GNU MPFR Library

Tomoya Michinaka[1,a)] Hideyuki Kawabata[1,b)] Tetsuo Hironaka[1,c)]

Received: February 16, 2021, Accepted: May 27, 2021

Abstract: The GNU MPFR library for arbitrary-precision floating-point arithmetic is widely used, and its Foreign Function Interface bindings to various languages have been developed. For the Rust programming language, existing bindings to the MPFR library include gmp-mpfr-sys (a low-level binding) and Rug (a binding that utilizes gmp-mpfr-sys to provide a more user-friendly interface). However, neither has sufficient descriptiveness and performance as bindings for general users of Rust, which is a programming language featuring high memory safety and high speed. We have developed a Rust binding, Rumpfr, to the MPFR library, that offers an easy way to write programs that perform high-speed multiple-precision floating-point computation. Rumpfr provides an interface that follows that of the MPFR library but hides the complexity of managing the mantissa area of floating-point numbers from the user. Rumpfr uses Rust's variable-length arrays to allocate mantissa areas, making it easy to handle without compromising Rust's high memory safety. In this paper, we describe the design and implementation of Rumpfr and present the results of numerical experiments demonstrating that Rumpfr can be used to write programs with low overhead.

Keywords: MPFR library, arbitrary-precision arithmetic, Rust bindings

1. Introduction

Scientific and technical computations by computer are generally accomplished by using floating-point representations of numerical values and floating-point operations on those values. Major general-purpose microprocessors are equipped with hardware capable of high-speed single- and double-precision floating-point arithmetic based on the IEEE 754 standard, and numerical types based on floating-point representation are treated as one of the basic types in many general-purpose programming languages. However, floating-point representation and floating-point arithmetic based on that representation often suffer from computational errors due to the approximation of numerical values on the basis of finite word lengths.

In floating-point representation, a real number is approximated by a triple of sign, mantissa, and exponent. Of these, the mantissa determines the upper limit on the number of significant digits of the numerical value obtained as a result of an arithmetic operation. To perform higher precision floating-point arithmetic, arbitrary-precision arithmetic can be used as it enables the length of the mantissa to be set arbitrarily. The GNU MPFR library (hereinafter referred to as MPFR) [1], [5] is widely used for arbitrary-precision numerical computation. Further, bindings for using it with various programming languages such as C++, Haskell, OCaml, Java, and Rust have been developed by third parties [4], [6], [12], [13], [14].

In MPFR, the mantissa of the floating-point representation is managed by a pair of an array of type `mp_limb_t` [*1] and the mantissa length to be used. Therefore, floating-point arithmetic using MPFR requires allocating and deallocating the mantissa of each floating-point number to and from the heap area, and the user is responsible for doing so appropriately. Bindings to languages with garbage collection, such as OCaml, Haskell, and Java, have been designed with interfaces that free users from the need to be aware of heap area management during programming. However, gmp-mpfr-sys [13] and Rug [12], which are bindings to Rust, do not have sufficient descriptiveness and performance as bindings for general users of Rust [2], a programming language that features both high memory safety and high speed.

We have developed **Rumpfr**, a Rust binding to MPFR that enables users to write efficient programs utilizing MPFR without the bother of heap area management. While using the standard variable-length array Vec<T> in Rust for the mantissa area, Rumpfr enables arithmetic operations with MPFR functions outside the Foreign Function Interface (FFI) boundary to be performed without causing problems. Unlike gmp-mpfr-sys, Rumpfr does not require the user to explicitly allocate or deallocate an area for the mantissa. In addition, unlike Rug, Rumpfr can be used to write efficient programs for matrix computation, a typical application in numerical computation, because the API design, which follows that of MPFR, makes it easy to reduce the frequency of memory allocation and data replication for multiple-precision floating-point numbers.

In this paper, we describe the design and implementation of Rumpfr. We also discuss the results of numerical experiments conducted to compare Rumpfr with gmp-mpfr-sys, Rug, and the

[1] Hiroshima City University, Hiroshima 731–3194, Japan
[a)] michinaka@ca.info.hiroshima-cu.ac.jp
[b)] kawabata@hiroshima-cu.ac.jp
[c)] hironaka@hiroshima-cu.ac.jp

[*1] Typically, a 64-bit unsigned integer.

case in which MPFR is driven by the C programming language.

This paper is organized as follows. In Section 2, we present the preliminaries: the necessity of multiple-precision floating-point operations, the MPFR library, and programming using MPFR. Section 3 presents the design of Rumpfr, and Section 4 presents the results of numerical experiments. In Section 5, we discuss the differences between Rumpfr and other bindings. Section 6 gives concluding remarks.

2. Preliminary: MPFR and Its Bindings

2.1 Necessity of Multiple-precision Floating-point Operations

Double-precision floating-point representation based on the IEEE 754 standard enables numerical values to be captured with a precision of 53 bits in terms of mantissa length, or about 16 digits in decimal notation. However, this precision is not necessarily sufficient for numerical simulation programs. Here is an example that illustrates the problem of computational errors in floating-point arithmetic [9]. Suppose that matrix A and vector b are as follows.

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

The solution of the linear system $Ax = b$ is 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, computation of Eqs. (1) and (2) using double-precision floating-point arithmetic gives values for x_1 and x_2 of 102558961.0 and 41869520.5, respectively. These values do not match the true solution for any single digit. This inaccuracy is difficult to identify immediately by looking at the Eqs. (1) and (2) or the calculated values.

The cause of these inaccurate computational results is the destructive loss of significant digits. Ideally, the loss of significant digits should be prevented by reformulating the equations or changing the computational procedure, but, in some cases, it can be easily prevented by simply using multiple-precision floating-point arithmetic with a longer mantissa. For example, in the above example, if we allocate 54 bits to the mantissa (making it one bit longer than needed for double precision), we can obtain values that match the true solution. It is difficult to determine in advance the mantissa length needed to obtain accurate results, but, by using arbitrary-precision arithmetic, we can examine the results by changing the length of the mantissa and estimate the degree of error introduced.

2.2 Multiple-precision Floating-point Library MPFR

The GNU MPFR library (MPFR) is a portable library for arbitrary-precision floating-point arithmetic written in the C programming language [5]. It provides functions for arithmetic operations with arbitrary mantissa lengths as well as basic mathematical functions. MPFR is still being actively developed 20 years after its first release [*2]. MPFR uses the GNU MP library [7] as its core. Although the basic interface is designed for the C language, it can be used with other languages such as C++, Haskell, OCaml, Java, and Rust by using the various FFI bindings that have been developed [4], [6], [12], [13], [14].

[*2] The latest version as of February 2021 is version 4.1.0, released in April 2020 [15].

```
typedef struct {
  mpfr_prec_t  _mpfr_prec;
  mpfr_sign_t  _mpfr_sign;
  mpfr_exp_t   _mpfr_exp;
  mp_limb_t    *_mpfr_d;
} __mpfr_struct;

typedef __mpfr_struct mpfr_t[1];
```

Fig. 1 Definition of floating point representation (`__mpfr_struct` structure) in MPFR.

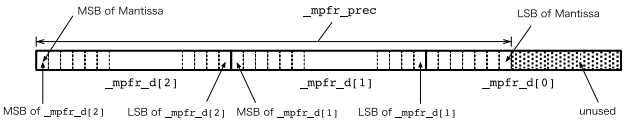

Fig. 2 Mantissa part of floating-point representation in MPFR (using arrays of type `mp_limb_t` in bit units).

```
1  #include <stdio.h>
2  #include <mpfr.h>
3
4  int main() {
5    mpfr_t x, y, z;
6    mpfr_init2(x, 200);
7    mpfr_set_d(x, 2.0, MPFR_RNDN);
8    mpfr_init2(y, 200);
9    mpfr_set_d(y, 7.0, MPFR_RNDN);
10   mpfr_init2(z, 200);
11   mpfr_div(z, x, y, MPFR_RNDN);
12   mpfr_printf("%.30Rf\n", z);
13   mpfr_clear(x);
14   mpfr_clear(y);
15   mpfr_clear(z);
16   mpfr_free_cache();
17   return 0;
18 }
```

Fig. 3 Example of a C program written using MPFR.

Floating-point numbers handled by MPFR have a fixed-length exponent part and an arbitrary-precision mantissa part. The MPFR library provides two functions (`mpfr_init2` and `mpfr_clean`) to manage the mantissa part of floating-point numbers.

Figure 1 shows the details of the `__mpfr_struct` type, which is the data type of numbers handled by the MPFR library [*3]. As shown in Fig. 1, the type `__mpfr_struct` is the structure for a floating-point representation with a mantissa part (`_mpfr_prec`), a sign (`_mpfr_sign`), an exponent part (`_mpfr_exp`), and a pointer to the mantissa part (`_mpfr_d`). The management of the memory area used for storing the mantissa part is left to the user. The bit arrangement of the area pointed to by the mantissa part _mpfr_d is shown in **Fig. 2**.

An example of a program written using MPFR is shown in **Fig. 3**. Areas sufficient for 200-bit mantissas are allocated in lines 6, 8, and 10. Each allocated area must be explicitly released, as shown in lines 13, 14, and 15.

[*3] In the MPFR user's manual [15], there is no reference to data of type `__struct_mpfr`. There is a reference only to data of type `mpfr_t`, another name for arrays of type `__struct_mpfr` (see Fig. 1). Although it is not usually necessary to know the internal structure of a floating-point number to use the MPFR library, the user should always be aware of whether a mantissa area has been allocated.

2.3 Java, OCaml, and Haskell Bindings to MPFR

mpfr-java [6] is a Java binding to MPFR that uses the Java Native Interface (JNI) via HawtJNI [8] to implement the MPFR binding. When using mpfr-java, in addition to using the same writing style as when MPFR is used in the C language, method chains can be used to write in a manner that resembles binary arithmetic notation with infix operators. The MPFR functions `mpfr_init2` and `mpfr_clear` are used to allocate and deallocate mantissa areas, but there is no need to include calls to these functions in the user program.

mlmpfr [14] is an OCaml binding to MPFR. The user does not need to allocate and deallocate space for floating-point arithmetic data. The space for storing the result of each floating-point operation is automatically allocated, and is not reused. The MPFR functions `mpfr_init2` and `mpfr_clear` are used internally to allocate and deallocate the mantissa area.

hmpfr [4] is a Haskell binding to MPFR. As with the OCaml binding, the user does not need to allocate and deallocate space for floating-point arithmetic data as the space for storing the result of each floating-point operation is automatically allocated. An interface using the ST monad is also provided to support reuse of the areas. hmpfr uses Haskell's `Foreign.ForeignPtr.mallocForeignPtrBytes` function instead of MPFR's functions for allocating and deallocating the mantissa area, and the heap area for the mantissa is managed by Haskell's garbage collector.

2.4 Existing Rust Bindings to MPFR

The programming language Rust [2] does not have a garbage collector; instead, it uses an ownership system to manage the heap area, thereby achieving both high speed and memory safety. It can be used as a system programming language to describe low-level processing and is thus expected to replace C++, which has been widely used as a programming language for applications that require high speed and high execution efficiency. An example of Rust's usefulness is its implementation of Servo [10], an engine for web browsers, that has achieved several times faster performance with less code than using C++ while avoiding the frequently occurring bugs of memory use after free [3].

Since Rust's high speed and memory safety are also useful for writing large-scale numerical programs, the development of efficient Rust bindings for MPFR is essential. However, the gmp-mpfr-sys Rust binding [13], a low-level library, requires users to explicitly allocate and deallocate mantissa areas using the functions `mpfr_init2` and `mpfr_clean` provided by MPFR. This is the same as with the direct use of MPFR in the C language. Most of the functions provided by gmp-mpfr-sys must be used in unsafe blocks. Thus, users are forced to do a difficult job to be sensitive to manage memory area while writing supposed-to-perform-efficiently programs not to, for example, let NULL pointer references to occur. An example of a program written using gmp-mpfr-sys is shown in **Fig. 4**. It is clear that more care is required in the description than when using MPFR in the C language.

On the other hand, the Rug Rust binding [12], which is based on gmp-mpfr-sys and provides a more user-friendly interface, enables simple descriptions of mathematical expressions by operator overloading, but it is difficult for users to write programs without being aware of the existence of memory areas that hold the calculation results of subexpressions.

```
use std::mem::MaybeUninit;
use gmp_mpfr_sys::mpfr::{self, rnd_t}

fn main(){
  unsafe{
    let rnd: rnd_t = rnd_t::RNDN;

    let mut x = MaybeUninit::uninit();
    mpfr::init2(x.as_mut_ptr(), 200);
    let mut x = x.assume_init();
    mpfr::set_d(&mut x, 1.234, rnd);

    let mut y = MaybeUninit::uninit();
    mpfr::init2(y.as_mut_ptr(), 200);
    let mut y = y.assume_init();
    mpfr::set_d(&mut y, 0.987654, rnd);

    let mut z = MaybeUninit::uninit();
    mpfr::init2(z.as_mut_ptr(), 200);
    let mut z = z.assume_init();

    mpfr::add(&mut z, &x, &y, rnd);

    mpfr::printf("%Rf=%Rf+%Rf\n\0".
        as_ptr() as *const c_char,
        &z, &x, &y);

    mpfr::clear(&mut x);
    mpfr::clear(&mut y);
    mpfr::clear(&mut z);
    mpfr::free_cashe();
  }
}
```

Fig. 4 Example Rust program written using gmp-mpfr-sys.

```
use rug::Float

fn main(){
  let x = Float::with_val(200, 1.234);
  let y = Float::with_val(200, 0.987654);
  let z = &x + y.clone();
  println!("{}={}+{}", z, x, y);
}
```

Fig. 5 Example Rust program written using Rug.

An example program written using Rug is shown in **Fig. 5**. Although the content of the program is the same as that of the program in Fig. 4, it requires less description. However, the arithmetic functions provided by Rug are based on the two-operand system, and one of the operands is always overwritten. Therefore, it is difficult to efficiently implement a program that refers to the results of many arithmetic operations as the operands of numerous operations, which is often seen in matrix computations. In addition, it is necessary to pay attention to the ownership of the operands used in the operations and to properly select between passing by reference and duplicating values using the `clone` method. As a result, even a short program such as that in Fig. 5 requires a less-than-intuitive description. The style of Rug's description, which does not require specifying an area for storing the result of an operation by operator overloading, tends to make programming that efficiently uses storage space rather difficult, especially in the case of implementing more complex numerical algorithms.

3. Rumpfr Rust Binding

3.1 Features of Rumpfer

The Rumpfr Rust binding to MPFR has the following features:

(1) It follows the interface of the functions provided by MPFR, facilitating its use by current MPFR users.
(2) As with MPFR, the allocated mantissa area can be reused, and overhead due to binding is kept low, making it possible to write efficient numerical programs.
(3) The user simply needs to be aware of the mantissa length when performing multiple-precision floating-point operations; numerical programs can be written without awareness of mantissa area management (allocation and deallocation).
(4) Unsafe operations are hidden from the user, eliminating the need for the user to write unsafe blocks.
(5) It is implemented using only the descriptions available in standard Rust without using Rust's external memory management mechanism.

None of the existing Rust bindings have all of these features. Although the last feature does not necessarily affect performance or usability, it is one of the advantages of Rumpfr whose implementation is highly independent of the outside of the Rust environment.

3.2 `Rfloat` Type: Floating Point Representation in Rumpfr

Figure 6 shows the definition of the `Rfloat` type, a floating-point number type used in Rumpfr. The `repr(C)` in the first line makes the Rfloat type compatible with a C structure, enabling it to be passed to MPFR functions as a `__struct_mpfr` structure.

We use the buffer area of Rust's variable-length array type Vec<`c_ulong`>[*4] for the mantissa part of Rumpfr's floating-point type. Applying the `as_mut_ptr()` method to an object of type Vec<T> yields a raw pointer to the buffer area in the object; the pointer is stored in the field `d` of type `Rfloat`. Since the buffer inside an object of type Vec<`c_ulong`> is simply an array of type `c_ulong`, it can be treated as an array of type `unsigned long` outside the FFI boundary under careful control.

It is important to note that Rust's memory management system does not traverse the locations pointed to by raw pointers. This means that even if a reference to a part of an object with a certain structure is still alive as a raw pointer, the entire area for that object is deallocated when control leaves the scope of the variable that owns the object. Therefore, simply embedding a raw pointer pointing to the buffer of Vec<T> into the `Rfloat` type does not enable the buffer to be used as a mantissa area in the MPFR functions.

In Rumpfr, a pointer to the object of type Vec<T> whose buffer is used as the mantissa part is also embedded in the data structure of type `Rfloat`. This (1) prevents the entire object of type Vec<T> from being released (including the buffer area) before invoking an MPFR function, and (2) allows the entire object of type Vec<T> to be released once the MPFR function has finished. In doing so, we convert a pointer to an object of type Vec<T> into a raw pointer type `*mut c_void` and embed it as shown in Fig. 6.

[*4] In general, it is the same as the Vec<u64> type.

```
1  #[repr(C)]
2  pub struct Rfloat {
3    prec:     c_long,
4    sign:     c_int,
5    exp:      c_long,
6    d:        *mut c_ulong,
7    dvec_ptr: *mut c_void,
8  }
```

Fig. 6 Definition of floating-point representation for Rumpfr (`Rfloat` structure).

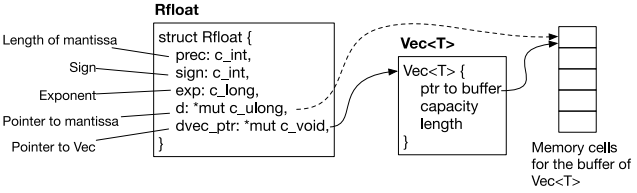

Fig. 7 Rumpfr's floating-point representation (type `Rfloat`) at runtime.

```
1  pub fn gen_Rfloat(len: u32) -> Rfloat {
2    if len == 0 { panic!("len == 0."); }
3    let bw   = size_of::<c_ulong>() as u32;
4    let dl   = (len - 1) / (bw * 8) + 1;
5    let dvec =
6          Vec::with_capacity(dl as usize)
7    unsafe { dvec.set_len(dl as usize); }
8    let mut dvec_ptr = Box::new(dvec);
9    let dp   = dvec_ptr.as_mut_ptr();
10   let vp   = Box::into_raw(dvec_ptr);
11   Rfloat {
12     prec:     len.into(),
13     sign:     1,
14     exp:      10,
15     d:        dp,
16     dvec_ptr: vp as *mut c_void,
17   }
18 }
```

Fig. 8 Constructor for Rumpfr floating-point representation of type `Rfloat`.

This conversion is required because it is inhibited (or treated as FFI-unsafe) to expose a Rust object outside FFI boundary if the object contains a non-raw pointer pointing to an object of type Vec<T>.

Figure 7 shows Rumpfr's floating point representation (type `Rfloat`) at runtime. The constructor for Rumpfr floating point representation of type `Rfloat` is shown in **Fig. 8**.

Passing a pointer pointing to an object as a raw pointer outside the FFI boundary prevents Rust from releasing the memory area unexpectedly, but memory safety cannot be maintained unless the processing outside the FFI boundary is valid. Thus, Rumpfr's design requires that the MPFR functions satisfy the following two points for valid operation.

- MPFR functions should never refer to an area that is not of type `__struct_mpfr`.
- MPFR functions should not access a memory area beyond the range defined by the mantissa length given by `prec`.

3.3 Implementation of Drop trait for `Rfloat` Type

The area pointed to by a raw pointer is temporarily unmanaged by Rust, and leaving it unmanaged will lead to memory leaks. The `Rfloat` type implements the Drop trait, and when an object of the `Rfloat` type is dropped (the memory area is released), the method `drop` puts the mantissa area to which it points under the control of Rust again, thereby preventing memory leaks. There

```
impl Drop for Rfloat {
 fn drop(&mut self) {
  unsafe {
    Box::from_raw(self.dvec_ptr
                  as *mut Vec<c_ulong>);
  }
 }
}
```

Fig. 9 Implementation of Drop trait for `Rfloat` type.

```
impl Clone for Rfloat {
  fn clone(&self) -> Rfloat {
    let vec = unsafe {
      Box::from_raw(self.dvec_ptr
                    as *mut Vec<c_ulong>)
    };
    let mut dvec_ptr =
              Box::new((*vec).clone());
    Box::into_raw(vec);
    let dp = dvec_ptr.as_mut_ptr();
    let vp = Box::into_raw(dvec_ptr);
    Rfloat {
      prec:     self.prec,
      sign:     self.sign,
      exp:      self.exp,
      d:        dp,
      dvec_ptr: vp as *mut c_void,
    }
  }
}
```

Fig. 10 Implementation of Clone trait for `Rfloat` type.

is thus no need for the user to explicitly deallocate the mantissa area.

The implementation of the Drop trait method `drop` is shown in **Fig. 9**. First, the ownership of the region pointed to by the raw pointer field `dvec_ptr` is retrieved to Rust, and then the region is dropped. During the dropping process, the object is re-recognized as Vec<`c_ulong`> type, and the buffer used as a mantissa area that is separately pointed to by the other raw pointer is released at the same time.

3.4 Implementation of Clone Trait for `Rfloat` Type

One of the most important applications of Rumpfr is writing matrix computation programs using multi-dimensional arrays. The most common way to handle arrays in Rust is to use the Vec<T> type, in which it is convenient and efficient to use the `vec!` macro to initialize the array data. To use the `vec!` macro with Vec<`Rfloat`>, the Clone trait must be implemented; i.e., the `clone` method must be implemented for deep copying. The implementation of the `clone` method in Rumpfr is shown in **Fig. 10**.

As described in Section 3.2, data of type `Rfloat` embeds a raw pointer pointing to the Vec<`c_ulong`> type that holds the mantissa. To deal with the object pointed to by the raw pointer and achieve deep copying appropriately, the `clone` method is implemented to clone the mantissa as Vec<`c_ulong`> by combining `Box::from_raw` and `Box::into_raw` and convert the pointer to a newly obtained Vec object into a raw pointer.

3.5 Interface to Floating-point Arithmetic Provided by Rumpfr

A user program written in Rust consists of a combination of calls to wrapper functions with public attributes as exemplified in **Fig. 11**. Since most wrapper functions simply call MPFR functions, the resulting overhead should be negligible.

```
extern "C" {
  #[link_name = "mpfr_add"]
  fn mpfr_add(r3: Rfloat_ptr,
              r1: Rfloat_constptr,
              r2: Rfloat_constptr,
              rnd: rnd_t) -> c_int;
  ...
}

pub fn add(r3: Rfloat_ptr,
           r1: Rfloat_constptr,
           r2: Rfloat_constptr,
           rnd: rnd_t) {
   unsafe { mpfr_add(r3, r1, r2, rnd) }
}
...
```

Fig. 11 Definitions of wrapper functions that call MPFR functions.

```
use ::rumpfr::rumpfr::{rnd_t, gen_Rfloat,
              set_str, add, printf3}

fn main(){
  let rnd = rnd_t::RNDN;

  let mut x = gen_Rfloat(200);
  let mut y = gen_Rfloat(200);
  let mut z = gen_Rfloat(200);

  set_str("1.2345", &mut x, rnd);
  set_str("0.987654", &mut y, rnd);
  add(&mut z, &x, &y, rnd);

  printf3("%Rf=%Rf+%Rf\n", &z, &x, &y);
}
```

Fig. 12 Example Rust program written using Rumpfr.

3.6 Programming with Rumpfr

An example program written using Rumpfr is shown in **Fig. 12**. Space for variables of type `Rfloat` is allocated in lines 7 to 9 of Fig. 12. The user can allocate a mantissa of the required length by using the constructor shown in Fig. 8 with the required length as an argument. Numbers are stored in two variables in lines 11 and 12, and the sum of the two numbers are calculated and stored in variable `z` in line 13. Unlike C programs that use MPFR, as shown in Fig. 3, and Rust programs that use gmp-mpfr-sys, as shown in Fig. 4, explicit deallocation of the mantissa area is not needed. In addition, unlike programs that use Rug as shown in Fig. 5, there is no need to worry about distinguishing between references and clones.

3.7 Alternative Implementation Strategy

In the design of Rumpfr described in Section 3.2, a buffer of type Vec<`c_ulong`> is used to store the mantissa. An object of type Vec<`c_ulong`> containing the buffer region is held during floating-point operations. It is also possible to extract and use only the array part of `c_ulong` from an object of type Vec<`c_ulong`>. This can be done by using a floating-point representation similar to MPFR's `__mpfr_struct`, as shown in **Fig. 13**, the constructor shown in **Fig. 14**, and the destructor shown in **Fig. 15**. The buffer of Vec<`c_ulong`> is used as the mantissa area also with this approach, but the buffer is extracted from Vec<`c_ulong`> by using `into_boxed_slice` when type `Rfloat` is created and the object of type Vec<`c_ulong`> is disassembled.

```
#[repr(C)]
pub struct Rfloat {
  prec: c_long,
  sign: c_int,
  exp:  c_long,
  d:    *mut c_ulong,
}
```

Fig. 13 Definition of `Rfloat` structure (alternative version).

```
pub fn gen_Rfloat(len: u32) -> Rfloat {
  if len == 0 { panic!("len == 0."); }
  let bw = size_of::<c_ulong>() as u32;
  let dl = (len - 1) / (bw * 8) + 1;
  let dvec: Vec<c_ulong> =
        Vec::with_capacity(dl as usize);
  unsafe { dvec.set_len(dl as usize); }
  let sl = dvec.into_boxed_slice();
  let dp = Box::into_raw(sl);
  Rfloat {
    prec: len.into(),
    sign: 1,
    exp:  10,
    d:    dp as *mut c_ulong,
  }
}
```

Fig. 14 Implementation of constructors of type `Rfloat` (alternative version).

```
impl Drop for Rfloat {
  fn drop(&mut self) {
    unsafe {
      Box::from_raw(self.d);
    }
  }
}
```

Fig. 15 Implementation of Drop trait for `Rfloat` type (alternative version).

With this approach, it is impossible to use methods for Vec<T> to prepare procedures such as reallocating mantissa parts; the coding may thus become cumbersome. In addition, implementing the Clone trait requires a means of duplicating an array of type `c_ulong`.

4. Evaluation

To evaluate the effectiveness of Rumpfr, we conducted experiments using several numerical calculation programs. We measured the execution times of programs using gmp-mpfr-sys and Rug, which are existing Rust bindings, and C programs written using MPFR as well as programs using Rumpfr.

The specifications of the environment used for the evaluation are shown in **Table 1**. The functions used to measure the execution time were `clock` from the standard library for C and `std::time::Instant` for Rust.

4.1 Iteration of Arithmetic Operations

The times required for the addition and multiplication iterations of two floating-point numbers were measured. In all cases of MPFR/C, gmp-mpfr-sys, Rug, and Rumpfr, the MPFR functions `mpfr_add` and `mpfr_mul` were called in all the programs tested. For each program, we measured the time required for 100,000 iterations with the mantissa length ranging from 100 to 1,000,000. The measured results are shown in **Fig. 16** and **Fig. 17**, in which the horizontal axis is the mantissa length and the vertical axis is the elapsed time in seconds. The shortest time for ten measurement iterations is plotted.

Table 1 Specifications of environment used for evaluation.

CPU	3.3 GHz Dual-core Intel Core i7
OS	macOS 10.15.7
Memory	16 GB
Rust compiler	rustc 1.50.0 (-C opt-level=3)
C compiler	Apple clang version 12.0.0 (-O)
MPFR library	MPFR 4.1.0
gmp-mpfr-sys	gmp-mpfr-sys 1.4.0
Rug	rug 1.11.0

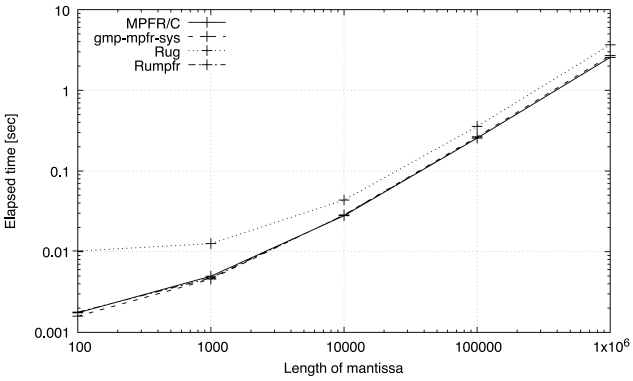

Fig. 16 Time required for 100,000 iterations of floating-point addition ($c \leftarrow a + b$).

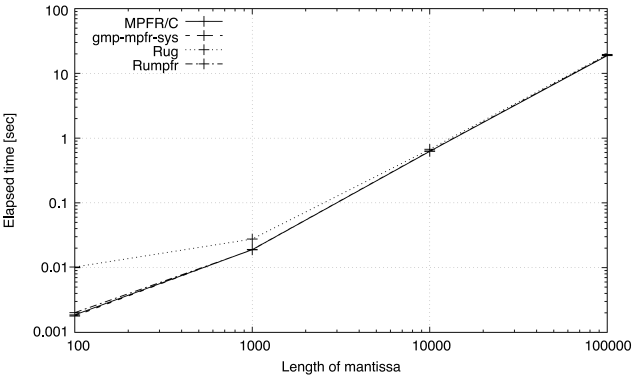

Fig. 17 Time required for 100,000 iterations of floating-point multiplication ($c \leftarrow a \times b$).

As can be seen from Fig. 16 and Fig. 17, the execution times for all the programs were about the same except for Rug, meaning that the performance degradation due to the use of binding was not significant. Basically, arithmetic operations in MPFR use a three-operand scheme in which the area for storing the computational result is allocated in advance, and the reference to it is passed to the arithmetic function. On the other hand, in Rug, arithmetic operations such as addition and multiplication are two-operand binary operations, which means that one of the floating-point numbers must be cloned (by performing a deep copy) in advance, and that incurs overhead for each operation [*5]. However, as can be seen from a comparison between Fig. 17 and Fig. 16, the overhead of allocating space for computing subexpressions is reasonably small when performing operations with high computational cost.

4.2 Solving Linear Equations

To investigate the performance of Rumpfr on typical numerical

[*5] Cumulative addition of $a \leftarrow a + b$ in Rug can be done without allocating a region or duplicating values by writing `a=a+&b`.

```
use ::rumpfr::rumpfr::{rnd_t, gen_Rfloat,
       set_str, set, add, sub, mul, div};
use std::time::Instant;
use std::env;

  ...

 let mut b = vec![gen_Rfloat(len); size];
 let mut m =
   vec![vec![gen_Rfloat(len);size];size];

  ...

 let rnd = rnd_t::RNDN;

 let mut t1 = gen_Rfloat(len);
 let mut t2 = gen_Rfloat(len);
 for k in 0 .. size {
  for i in k+1 .. size {
   div(&mut t2, &m[i][k], &m[k][k], rnd);
   set(&mut m[i][k], &t1l2, rnd);
   for j in k+1 .. size {
     mul(&mut t1, &m[i][k], &m[k][j], rnd);
     sub(&mut t2, &m[i][j], &t1, rnd);
     set(&mut m[i][j], &t2, rnd);
   }
  }
 }

 for i in 1 .. size {
  for j in 0 ..= i -1 {
   mul(&mut t1, &b[j], &m[i][j], rnd);
   sub(&mut t2, &b[i], &t1, rnd);
   set(&mut b[i], &t2, rnd);
  }
 }

 for i in (0 .. size).rev() {
  for j in (i+1 .. size).rev() {
   mul(&mut t1, &b[j], &m[i][j], rnd);
   sub(&mut t2, &b[i], &t1, rnd);
   set(&mut b[i], &t2, rnd);
  }
  div(&mut t2, &b[i], &m[i][j], rnd);
  set(&mut b[i], &t2, rnd);
 }

   ...
```

Fig. 18 Solving simultaneous linear equations by LU decomposition using Rumpfr.

programs, we measured the time required to solve a linear system of equations with dense coefficients by LU decomposition without pivoting [*6]. For each of the cases of MPFR, gmp-mpfr-sys, Rug, and Rumpfr, we measured the time required to allocate and initialize the array data and the time from the start to the end of the computation for various combinations of the number of unknowns and the precision used in the computation (i.e., the mantissa length). We ran each program multiple times, and then used the shortest execution time. The outline of the Rumpfr program used in the measurements is shown in **Fig. 18**.

The measured results when the number of unknowns was 100, 200, and 400 are shown in **Fig. 19**, **Fig. 20**, and **Fig. 21**, respectively. The vertical axis shows the elapsed time in seconds. The bar graphs are shown for each program for mantissa lengths of 250, 500, 1,000, 2,000, and 4,000 bits. Each bar has two parts: the top one shows the time required to allocate space before starting computation, and the bottom one shows the time required for computation.

We can observe the following things from the results shown in Fig. 19, Fig. 20, and Fig. 21.

- The computational cost increases about eight times when the number of unknowns is doubled, indicating that the validity of the program behavior for solving linear equations is not disturbed by the use of each binding.
- Compared with the C programs, which directly use the MPFR library, the Rust programs, which use gmp-mpfr-sys and Rumpfr, can be processed with sufficiently small overhead. In particular, when the mantissa length used in the floating-point arithmetic is large, the difference in overhead is small. One possible reason for the overhead of Rumpfr when the mantissa length is small is that it requires more memory than gmp-mpfr-sys for storing each floating-point number. A detailed analysis of this finding remains for future work.
- Among the three Rust bindings, the overhead of Rug stands out. This is because, as described in Section 2.4, binary op-

[*6] We used the Hilbert coefficient matrix as it does not break a calculation without pivoting.

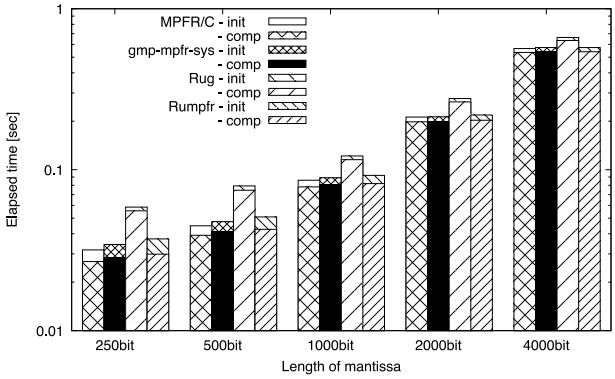

Fig. 19 Measured results of solving simultaneous linear equations (number of unknowns: 100).

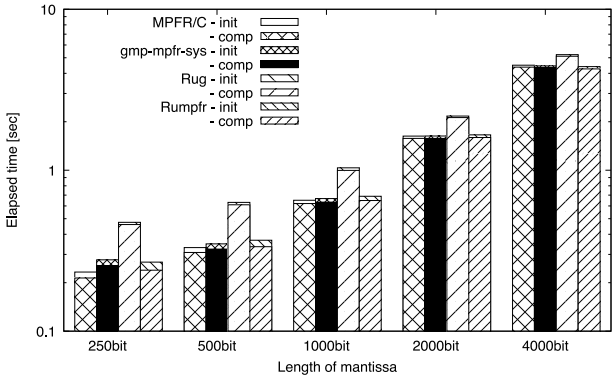

Fig. 20 Measured results of solving simultaneous linear equations (number of unknowns: 200).

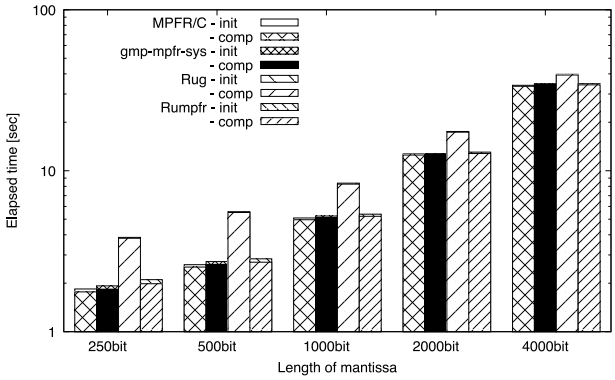

Fig. 21 Measured results of solving simultaneous linear equations (number of unknowns: 400).

erations in Rug frequently allocate numerical data areas and replicate data in order to deal with the fact that one operand area is overwritten. Rumpfr is faster than Rug and comparable to gmp-mpfr-sys, mainly because it follows the API of MPFR, which is based on frequent patterns in numerical computation.

5. Discussion

5.1 Current Status of Rumpfr

Rumpfr offers wrappers for the basic arithmetic operations and mathematical functions provided by the MPFR library and has sufficient functionality for writing general numerical programs [*7]. However, Rumpfr does not cover all the features provided by the MPFR library. For example, the MPFR library provides a way to change the mantissa size from the user program, but Rumpfr does not provide a way to expand the mantissa area of a floating-point number at runtime. In Rumpfr's floating-point representation, the mantissa area is a Rust dynamic array (of type Vec<T>), so it would be easy to provide a way to change the mantissa area in Rust.

Rumpfr is aimed at achieving a high level of both ease of writing and efficiency of program execution, under the assumption that users can directly write numerical programs composed of multiple-precision floating-point operations. On the other hand, gmp-mpfr-sys provides a low-level Rust binding to MPFR and is considered to be a means for establishing easy-to-use user interfaces such as those represented by Rug. While gmp-mpfr-sys enables the user to do everything that can be done with MPFR in C in Rust programs, Rumpfr is designed to enable efficient use of multiple-precision arithmetic with natural programming in Rust. Despite this difference in design philosophy, our experiments demonstrated that Rumpfr, as well as gmp-mpfr-sys, is useful for developing programs that implement general numerical algorithms that can be executed at high speed with low overhead.

The current version of Rumpfr was designed to be a Rust binding with the primary goal of making the MPFR library easier to use with Rust. In contrast, Rug, an existing Rust binding, has additional functionalities for handling multiple-precision integers and complex numbers. Although we used Rug as a target for comparison, since Rug is a binding that provides users with a wider variety of features, it is not possible to simply determine which is more beneficial.

5.2 Concerns About Loss of Memory Safety due to Use of Rumpfr

As described in Section 3.2, Rumpfr expects data of type `__struct_mpfr` and the array regions of a `unsigned long` mantissa to be properly handled outside the FFI boundary. In Rumpfr, when a floating-point type instance is created using `gen_Rfloat`, the mantissa area is prepared at the same time. Thus the NULL pointer is not exposed outside the FFI boundary [*8]. As long as the interface provided by Rumpfr is used in user programs, memory safety is not considered to be threatened.

6. Conclusion

We have developed Rumpfr, a binding for using MPFR from Rust in a fast and easy-to-use manner. By using Rumpfr, programs for multiple-precision floating-point arithmetic with MPFR can be written efficiently in Rust. In this paper, we describe the design and implementation of Rumpfr and discuss its usefulness based on numerical experiments.

Future work includes detailed performance evaluation and tuning using a variety of numerical applications. In addition, we plan to improve the functionality of the binding by, for example, developing interfaces for multiple-precision integers, complex numbers, and operations using them, as in Rug, and by introducing functions to make it easier to use libraries such as the MPFI interval arithmetic library [11] in Rust.

Acknowledgments The authors would like to thank Tasuku Hiraishi for his helpful comments.

References

[1] The GNU MPFR Library (online), available from ⟨https://www.mpfr.org⟩ (accessed 2021-02-18).
[2] The Rust Language (online), available from ⟨http://www.rust-lang.org/⟩ (accessed 2021-02-18).
[3] Anderson, B., Bergstrom, L., Goregaokar, M., Matthews, J., McAllister, K., Moffitt, J. and Sapin, S.: Engineering the Servo Web Browser Engine Using Rust, *Proc. 38th International Conference on Software Engineering Companion*, ICSE '16, Association for Computing Machinery, pp.81–89 (online), DOI: 10.1145/2889160.2889229 (2016).
[4] Bizjak, A. and Konečný, M.: hmpfr: Haskell binding to the MPFR library (online), available from ⟨http://hackage.haskell.org/package/hmpfr⟩ (accessed 2021-02-18).
[5] Fousse, L., Hanrot, G., Lefèvre, V., Pélissier, P. and Zimmermann, P.: MPFR: A Multiple-Precision Binary Floating-Point Library with Correct Rounding, *ACM Trans. Math. Softw.*, Vol.33, No.2, pp.13–es (online), DOI: 10.1145/1236463.1236468 (2007).
[6] K framework: GNU MPFR Java Bindings (online), available from ⟨https://github.com/kframework/mpfr-java⟩ (accessed 2021-02-18).
[7] Free Software Foundation: The GNU Multiple Precision Arithmetic Library (online), available from ⟨https://gmplib.org⟩ (accessed 2021-02-18).
[8] FuseSource: HawtJNI (online), available from ⟨https://github.com/fusesource/hawtjni⟩ (accessed 2021-02-18).
[9] Krämer, W.: A Priori Worst Case Error Bounds for Floating-Point Computations, *IEEE Trans. Computers*, Vol.47, No.7, pp.750–756 (1998).
[10] Project, T.S.: Servo (online), available from ⟨https://servo.org⟩ (accessed 2021-02-18).
[11] Revol, N.: MPFI, a multiple precision interval arithmetic library based on MPFR (online), available from ⟨https://perso.ens-lyon.fr/nathalie.revol/software.html⟩ (accessed 2021-02-18).
[12] Spiteri, T.: Arbitrary-precision numbers (online), available from ⟨https://crates.io/crates/rug⟩ (accessed 2021-02-18).
[13] Spiteri, T.: Rust low-level bindings for GMP, MPFR and MPC (online), available from ⟨https://crates.io/crates/gmp-mpfr-sys⟩ (accessed 2021-02-18).
[14] Thévenoux, L.: OCaml C bindings for MPFR-4.1.0 (online), available from ⟨https://opam.ocaml.org/packages/mlmpfr/⟩ (accessed 2021-02-18).
[15] The MPFR team: *GNU MPFR: The Multiple Precision Floating-Point Reliable Library*, 4.1.0 edition (2020).

[*7] In addition to the basic arithmetic operations, basic mathematical functions such as sign reversal, square root, and absolute value calculations, trigonometric and logarithmic functions are implemented. Numerical data can be generated from strings, duplicated, and converted into strings. Constraints on the mantissa, exponent, rounding mode, etc. follow MPFR.

[*8] It is assumed that runtime memory allocation by `Vec::with_capacity`, `Vec::clone()`, etc. does not fail.

Tomoya Michinaka received a B.E. degree in computer engineering from Hiroshima City University in 2021. His research interests include programming languages and systems.

Hideyuki Kawabata received B.E. and Ph.D. degrees in computer engineering from Kyoto University in 1992 and 2004, respectively. He is an associate professor at Hiroshima City University. His research interests include numerical programming and programming languages. He is a member of ACM, IEEE Computer Society, IPSJ, IEICE, JSIAM, and JSSST.

Tetsuo Hironaka received a Ph.D. degree in computer engineering from Kyushu University in 1993. From 1993 to 1994, he served as a research associate at Kyushu University. From 1994 to 2006, he was an associate professor at Hiroshima City University. Since 2006, he has been a professor at Hiroshima City University. His research interests include computer architectures, reconfigurable architectures, and software engineering. He is a member of IPSJ, IEICE, IEEE, and ACM.