Generic rings in FLINT

Fredrik Johansson

Abstract We describe GR, a system for type-generic algebraic computation implemented in the C library FLINT. We combine lightweight dynamic dispatch with high-performance mutable data structures, optimized kernel operations and generic high-level algorithms. We offer robust support for inexact ring implementations such as real and complex numbers represented using ball arithmetic.

1 Introduction

One of the jobs of a computer algebra system is to support generic programming, allowing a single algorithm to be executed for different types of input. Different types can represent different algebraic structures, or perhaps different implementations of the same algebraic structure (large or small integers; exact or approximate real numbers; dense or sparse polynomials). In a well-designed generics system, one should be able to express the concerns of high-level mathematical algorithms orthogonally to the low-level implementation details and performance tradeoffs of specific types [21].

There are many existing libraries combining some generic model for mathematical objects with techniques for high-performance computing, offering different tradeoffs between speed, dynamicity, type safety and other aspects. For a non-exhaustive list, we mention SageMath [28], Magma [6], Oscar [10], GAP [3], Pari/GP [27], CoCoA [1], Singular [11], LinBox [12], Mathemagix [29] and BPAS [7]. Some systems use a small C/C++ kernel for efficient arithmetic on some restricted set of types (e.g. GMP [16] for variable-size integers), leaving the generic constructions to a higher-level language; others implement a sophisticated mathematical object system with generic programming support directly in C or C++ or a similar low-level language.

In this work, we present GR ("generic rings"), a generics system in C for the FLINT library [14]. GR was introduced in FLINT 3.0 in 2023 and is still under active develop-

Fredrik Johansson
Inria, IMB (UMR 5251), e-mail: fredrik.johansson@gmail.com

ment; features described in this article refer to FLINT 3.3. Documentation is available at `https://flintlib.org/doc/gr.html`.

FLINT provides many specific C types corresponding to different algebraic structures, mostly rings. Until recently, FLINT used some ad hoc solutions for limited internal generic programming but lacked a general-purpose generics system. We created GR in part to be able to more easily add new features to FLINT itself, in part to be able to refactor FLINT's internals for code size and performance, in part to address shortcomings in computer algebra systems like SageMath, Oscar and SymPy [26] which use FLINT as a backend library.

While many implementation techniques in GR are standard, we believe it is worthwhile to document our specific design choices by which we attempt to simultaneously maximize runtime flexibility, runtime performance, and compile-time efficiency.

A notable functionality of GR is that we support inexact representations in the mathematically rigorous sense of interval arithmetic. This feature is motivated by analytic applications requiring different models of real and complex numbers, but is applicable more generally, e.g. to exact rings with symbolic parameters or undecidable predicates.

This article is intended in part as an introduction to GR for prospective users, in part as an implementation case study which may be instructive for developers of other mathematical libraries.

2 Feature overview

GR allows constructing context objects defining implementations of various algebraic structures and provides a generic interface for manipulating elements of such structures.

Context constructors are provided for working with most of FLINT's builtin types: rational numbers represented by `fmpq`, number field elements represented by `nf_elem`, polynomials in $\mathbb{Q}[X]$ represented by `fmpq_poly`, real numbers represented by `arb` balls, etc. The context object stores eventual ring parameters (such as the modulus for a residue ring) and computational parameters (such as precision).

GR offers a wide range of methods for basic arithmetic, special functions, polynomial arithmetic, linear algebra, etc., largely encompassing the functionality previously provided in FLINT for each individual type. Since GR provides a single, uniform interface to a large part of FLINT's more than 100 C types and 10,000 C functions, it notably simplifies the job for other software to interface with FLINT.

In addition, GR provides the following parametric types:

- `gr_vec` - dense vectors - $\mathrm{Vec}(R)$ (variable length) or $\mathrm{Vec}(R, n)$
- `gr_mat` - dense matrices - $\mathrm{Mat}(R)$ or $\mathrm{Mat}(R, m, n)$
- `gr_poly` - dense univariate polynomials - $R[X]$
- `gr_mpoly` - sparse multivariate polynomials - $R[X_1, \ldots, X_n]$
- `gr_series` - power series - $R[[X]]$
- `gr_series_mod` - truncated power series - $R[[X]]/\langle X^N \rangle$
- `gr_fraction` - fraction fields - $\mathrm{Frac}(R)$

These types are fully recursive and can use builtin FLINT types, other generic types or user-defined types for the ground domain R. The system is completely dynamic: the user may define their own types at runtime. Planned future extensions include generic sparse vectors, sparse matrices, and Ore polynomials.

3 Basic data structures and interface

We use a typical pointer-based approach to generic programming in C: a ring R is represented by a context object (of type `gr_ctx_t`) containing the following data:

- the element size in bytes, i.e. `sizeof(T)` where `T` is the C type of an element,
- a method table (an array of C function pointers),
- a numerical code identifying the context object for internal use.

Some fundamental methods for memory management, conversions and arithmetic must be implemented by each context object; other methods have generic default implementation which a context object can optionally override with more performant versions.

Elements are stored in mutable variables which must be initialized before use and cleared after use by calling the methods `gr_init` and `gr_clear`. References to elements are passed around as `void` pointers. We define `gr_ptr` as a type alias for a pointer to a writable (output) variable and `gr_srcptr` for a pointer to an input variable which should be read-only. These aliases are also used for pointers to arrays of contiguous elements.

The programming interface can be illustrated with the following code example which defines a function to compute $\operatorname{dot}(u, v) = \sum_{i=0}^{n-1} u_i v_i$, constructs the ring of Gaussian integers $\mathbb{Z}[i]$ (with elements represented by the builtin FLINT type `fmpzi`), constructs the polynomial ring $\mathbb{Z}[i][x]$, creates the vector $u = [x+i, x-i]$, computes $s \leftarrow \operatorname{dot}(u, u)$ $(= -2 + 2x^2)$, and prints s.

GR code example

```
#include "flint/gr.h"

int
dot(gr_ptr res, gr_srcptr u, gr_srcptr v, slong n, gr_ctx_t ctx)
{
    int status = GR_SUCCESS;
    slong i, sz = ctx->sizeof_elem;
    gr_ptr t;
    GR_TMP_INIT(t, ctx);

    status = gr_zero(res, ctx);
    for (i = 0; i < n; i++)
    {
        status |= gr_mul(t, GR_ENTRY(u, i, sz), GR_ENTRY(v, i, sz), ctx);
        status |= gr_add(res, res, t, ctx);
```

```
    }

    GR_TMP_CLEAR(t, ctx);
    return status;
}

int main()
{
    gr_ctx_t ZZi, ZZix;
    gr_ptr u, s;
    int status = GR_SUCCESS;

    gr_ctx_init_fmpzi(ZZi);
    gr_ctx_init_gr_poly(ZZix, ZZi);
    GR_TMP_INIT_VEC(u, 2, ZZix);
    GR_TMP_INIT(s, ZZix);

    status |= gr_set_str(GR_ENTRY(u, 0, ZZix->sizeof_elem), "x+i", ZZix);
    status |= gr_set_str(GR_ENTRY(u, 1, ZZix->sizeof_elem), "x-i", ZZix);
    status |= dot(s, u, u, 2, ZZix);
    status |= gr_println(s, ZZix);    // Outputs -2 + 2*x^2

    GR_TMP_CLEAR_VEC(u, 2, ZZix);
    GR_TMP_CLEAR(s, ZZix);
    gr_ctx_clear(ZZix);
    gr_ctx_clear(ZZi);

    return status;
}
```

Here, `gr_zero`, `gr_mul` and `gr_add` are inline C functions which look up and call the implementations of the relevant operations in the method table of `ctx`. We note that these methods must be passed the context object and the input operands as well as an output variable which is mutated in-place.

The `GR_ENTRY` macro is used to access the ith element of a vector with elements `sz` bytes apart. The status codes used for error handling are discussed below in section 8.

The `GR_TMP_INIT` macro allocates a temporary variable on the C stack and initializes it by calling `gr_init`. `GR_TMP_CLEAR` performs the corresponding cleanup. A stack variable in C can be used locally in a function and forwarded in subroutine calls, but cannot be returned up through the call stack. We mention that there are separate functions for heap-allocating variables, but they are rarely needed as most GR functions follow the convention that output variables are preallocated by the caller. The `GR_TMP_INIT_VEC` macro creates a temporary array of elements which will be placed on the stack or on the heap depending on the size.

4 Type system

Input and output operands in GR must have precisely the type specified by the context object. We do not extend the domain of results automatically; for example, division of elements in $\mathbb{Z}$ using the standard `gr_div` method produces exact quotients in $\mathbb{Z}$ rather than fractions in $\mathbb{Q}$ (see section 8 regarding error handling). If we want to generate fractions, we must explicitly convert our `fmpz` operands to new variables belonging to an `fmpq` rational number context.

We make this deliberate restriction in part for performance reasons as it allows dispensing with runtime type checks, but also because it removes a host of extension- and coercion-related ambiguities present in some computer algebra systems.

Coercions are possible by using mixed-type methods for which the user must specify the result type explicitly. For example, the function

```
int gr_mul_other(gr_ptr res, gr_srcptr x,
    gr_srcptr y, gr_ctx_t y_ctx, gr_ctx_t ctx)
```

allows computing $res \leftarrow x \cdot y$ where res, $x \in \mathbb{Q}$ and $y \in \mathbb{Z}$, and similarly for many other type combinations such as res, $x \in R[X]$ and $y \in R$. There is an analogous function `gr_other_mul` where the type of x rather than y differs from res, useful for noncommutative rings.

In many applications, it is crucial to be able to choose different algorithms or parameters depending on the properties of the algebraic structure one is dealing with. We do not attempt to introduce any class hierarchy for GR types. Instead, context objects overload predicate methods such as the following:

- `gr_ctx_is_commutative_ring`
- `gr_ctx_is_integral_domain`
- `gr_ctx_is_field`

These methods are expected to be cheap so that they can be used for efficient internal algorithm dispatch. For example, implementations of $R = \mathbb{Z}/m\mathbb{Z}$ should not test m for primality each time we check whether R is a field; instead, we allow setting a primality flag in the context objects for such types, and the predicate methods simply read this flag.

5 Performance and vectorization

The GR memory model makes the following key design choices for performance:

- Elements have the same type and do not require a header (all metadata is stored in the separate context object) and can thus be packed contiguously in vectors.
- Variables are mutable.

These choices help ensure compatibility between GR generics and non-generic FLINT code. For example, a polynomial with `fmpz` coefficients can be manipulated just as well using generic `gr_poly` methods as with specialized `fmpz_poly` methods.

GR supports both "shallow" (i.e. pointer-free) data structures and "deep" data structures (which may allocate variable-size data on the heap) for elements. An example of a deep type is GMP's arbitrary-size integer type `mpz` whose data structure contains a count of used words, a count of allocated words, and a pointer to a heap-allocated array of words. The context object for a deep type is expected to define a `gr_clear` method which deallocates data; for a shallow type, this destructor usually does nothing.

The use of mutable variables in GR is inspired by GMP's interface for `mpz` and helps amortize the cost of allocations, deallocations and pointer management in deep types, but cannot eliminate this overhead completely. FLINT's `fmpz` integer type attempts to achieve some of the benefits of a shallow type by using single-word data structure which can hold a small value $|n| < 2^{62}$ inline, turning into a pointer to a heap-allocated `mpz` only when needed to store a larger value.

A single GR operation like `gr_add` requires a method table lookup and a C function call, which together may have an overhead of a handful of CPU cycles (around one nanosecond). This is negligible for operations on complex objects like rational numbers or number field elements, but it is significant for individual operations on shallow machine types. To minimize the impact of runtime dispatch, we attempt to rely as much as possible on fused, vectorized and batched operations in high-level algorithms. GR methods can essentially be categorized into three levels:

- Elementwise operations like $z \leftarrow x + y$, which must be implemented by each ring.
- Vector operations like $z_i \leftarrow x_i + y_i$ or $y_i \leftarrow y_i \pm x_i \cdot c$. These have generic default implementations which call the corresponding elementwise operations in a loop.
- Higher-complexity operations (e.g. polynomial multiplication, matrix multiplication). These have generic default implementations which repeatedly call vector methods and other higher-complexity methods.

Ring implementations can optionally overload specific vector operations and/or higher-complexity operations with non-generic variants to improve performance. Optimizing vector operations for a specific type can have several benefits:

- Eliminating inner function calls.
- Allowing SIMD vectorization.
- Taking advantage of delayed reduction or preconditioning. The most important vector operations are dot products which greatly benefit from delayed modular reduction in residue rings and delayed sum normalization for multiprecision floating-point types [19].

Higher-complexity operations can further benefit from improved memory locality and use of asymptotically fast algorithms (e.g. FFT and Strassen multiplication).

Table 1 compares overheads for adding integer vectors of type `fmpz`. We observe that using the GR interface for elementwise operations instead of the non-generic `fmpz` interface incurs roughly a 30% overhead for small (single-word) integers and a 10% overhead for integers with a few words. The `_fmpz_vec_add` method is specially optimized for small coefficients, running up to three times faster than the elementwise loop.

Table 1 Time (nanoseconds) to add two length-n vectors over $\mathbb{Z}$.

Code	20-bit coefficients			200-bit coefficients	
	$n = 1$	$n = 10$	$n = 100$	$n = 1$	$n = 10$
`for (i = 0; i < n; i++)` `  fmpz_add(z + i, x + i, y + i);`	2.7	25.1	253.0	10.1	94.8

`for (i = 0; i < n; i++)`
`  status |= gr_add(`
`  GR_ENTRY(z, i, sz),`
`  GR_ENTRY(x, i, sz),`
`  GR_ENTRY(y, i, sz), ctx);`
	3.9	34.3	344.0	11.2	107.0

`_fmpz_vec_add(z, x, y, n);`	3.2	12.7	98.0	12.7	95.9
`status |= _gr_vec_add(z,` `  x, y, n, ctx);`	3.6	13.2	98.1	12.7	94.1

There is virtually no difference in performance between using the generic `_gr_vec_add` interface and the non-generic `_fmpz_vec_add` interface to invoke this vector operation.

We have replaced hundreds of previously non-generic functions in FLINT with wrappers around generic GR algorithms. In each case, we have carefully verified that this caused no measurable performance regressions once the relevant underlying vector operations had appropriate specializations in GR.

6 Specialization for parametric types

When a ring depends on a parameter, say $R = R_p$, and the size of elements can be bounded as a function of p, a GR implementation of R is free to choose the element size at runtime when the context object is created. This has two benefits: one may be able to use a shallow type instead of a deep type, and one can choose methods optimized for the specific parameter p.

There is no hard upper limit on the size of GR elements, but generally an element should not be larger than a few kilobytes to avoid the risk of stack overflow when allocating temporary variables on the C stack; elements larger than a few dozen words should therefore use heap allocation. (Indeed, for multi-kilobyte elements, allocation overheads are in any case generally going to be negligible.)

Historically, FLINT had only two representations of $\mathbb{Z}/m\mathbb{Z}$: the shallow `ulong` type for single-word m and `fmpz` for arbitrary-size m. After adding GR to FLINT, we implemented a new intermediate `mpn_mod` representation for $\mathbb{Z}/m\mathbb{Z}$ with $2^{64} \leq m \leq 2^{1024}-1$, where each element is stored shallowly using the optimal number $2 \leq \ell \leq 16$ of 64-bit words.

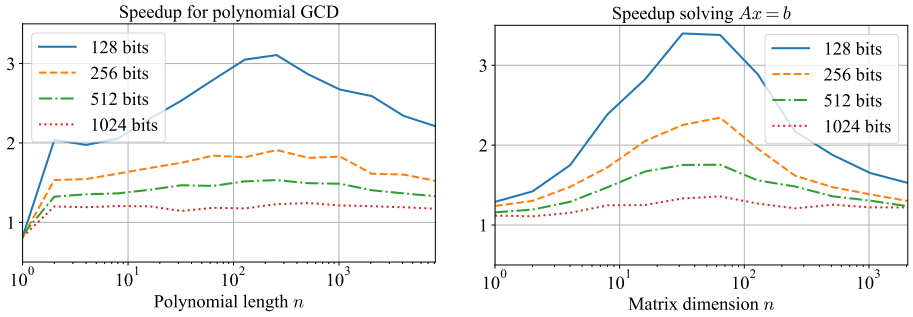

Fig. 1 Speedup using the `mpn_mod` representation for $\mathbb{Z}/m\mathbb{Z}$ arithmetic instead of the general-purpose `fmpz` integer format, for various sizes of m between 128 and 1024 bits.

Figure 1 illustrates the speedup of using `mpn_mod` instead of `fmpz` for polynomial GCD (`gr_poly_gcd`) and linear system solving (`gr_mat_nonsingular_solve`) over $\mathbb{Z}/m\mathbb{Z}$ for varying bit sizes of m and polynomial lengths or matrix sizes n.

For small n, these functions use basecase algorithms (the Euclidean algorithm and Gaussian elimination respectively), which are implemented using delayed modular reductions modulo m both in the case of `mpn_mod` and `fmpz`. For large n, both use generic GR implementations of standard divide-and-conquer methods (the half-GCD algorithm and block recursive LU factorization respectively).

Although `mpn_mod` and `fmpz` perform comparably when $n \to \infty$ as fast polynomial or matrix multiplication kicks in (the huge-n multiplication backends are the same for both formats), `mpn_mod` achieves a significant speedup for moderate values of n. This can be attributed to three factors:

- Faster arithmetic in the basecase algorithms.
- Less overhead for temporary allocations in the divide-and-conquer algorithms.
- Faster medium-sized polynomial and matrix multiplication, due to significantly cheaper additions and subtractions allowing the use of dedicated medium-size multiplication algorithms (Karatsuba and Waksman [30] multiplication) which rely heavily on trading multiplications for additions.

We have analogously implemented shallow real and complex floating-point formats (`nfloat`, `nfloat_complex`) with $1 \le \ell \le 66$ words of mantissa, permiting precisions between 64 and 4224 bits (in any multiple of 64). We note that this is equivalent to supporting 132 distinct C types `nfloat64`, `nfloat128`, ...). The improvement compared to MPFR's arbitrary-precision `mpfr` [15] or FLINT's `arf` [18] are similar to those reported for $\mathbb{Z}/m\mathbb{Z}$ above, with speedups of order $2\times$ on high-level benchmarks like polynomial root-finding and approximate linear solving over $\mathbb{R}$ and $\mathbb{C}$. In future work, we intend to extend these results to ball arithmetic.

In these parametric types, we currently do not generate a different method table for each parameter value, although this would be a possibility. However, the arithmetic op-

erations do switch internally between different fixed-length assembly routines, notably for multi-word integer multiplication [2].

We conclude that it is useful to generate specialized representations for different parameter values instead of using a one-size-fits-all type.

This is of course well known practice, notably exploited by several mathematical libraries using C++ templates for generics (ahead-of-time compilation) and more recently by libraries based on Julia [5] (with just-in-time compilation). One common drawback of such systems is that the whole program depending on R_p typically will be recompiled for each instance of p, which can make compile times a significant bottleneck; see [13] for a discussion of how just-in-time compilation costs prevent taking advantage of the full flexibility of Julia's type system for computer algebra. In contrast, our approach in GR is to selectively optimize only performance-critical kernel operations while using reasonably efficient runtime dispatch for everything else. We believe that this can achieve near-optimal runtime performance while leaving it feasible to have large programs with hundreds or thousands of parameter specializations.

Making it cheap to construct context objects in GR is important for instance in the support of multimodular algorithms, which may need to create a million instances of a ring like $\mathbb{Z}/m\mathbb{Z}$ or $(\mathbb{Z}/m\mathbb{Z})[X]$. Most context object initializers in GR perform no memory allocations or substantial precomputations, allowing for initialization time measured in single nanoseconds with a cached method table (constructing a method table, which only needs to be done once for a given type, costs roughly half a microsecond). Certain context constructors such as those for number fields [17] perform more expensive precomputations (perhaps comparable in cost to a handful of arithmetic operations in the ring) as a tradeoff to allow faster arithmetic.

A quantitative comparison of tradeoffs in GR and other generics implementations for parametric types would be a welcome subject for future study.

7 Inexact elements

GR supports inexact representations, adopting the semantics of interval arithmetic: a variable x is understood to represent an enclosure of possible element values, and methods are required to preserve inclusions. The interface is the same for inexact types and exact types; in the latter case, enclosures are simply singletons.

Builtin FLINT types with inexact representation include real and complex balls (`arb` and `acb`) and generic finite-precision power series (`gr_series`) where an $O(X^n)$ term represents an enclosure of series tails. Indeed, we can without problem construct an object like

$$[1 \pm 0.001] + [0 \pm 0.0001]X + O(X^2) \in \mathbb{R}[[X]]$$

mixing the two kinds of error bound.

All generic data structures and algorithms (e.g. for linear algebra) have been implemented from the ground up to work correctly with inexact elements. This is a notable improvement over several existing generic mathematical libraries and computer algebra

systems which often work unreliably with inexact types. A common problem is that an algorithm implements a conditional branch

```
if (x == 0)
    // handle the case x == 0
else
    // handle the case x != 0
```

with exact objects in mind, giving incorrect results when the predicate $x = 0$ evaluates to a false positive or a false negative given an inexact representation of x.

We get around this problem by using triple-valued logic for all predicates, i.e. by using enclosure semantics for boolean values (represented by the type `truth_t` in GR) and implementing all control flow from the ground up to deal correctly with the unknown case. Thus the above example generally translates to something like

```
truth_t zero = gr_is_zero(x, ctx);

if (zero == T_TRUE)
    // handle the case x == 0
else if (zero == T_FALSE)
    // handle the case x != 0
else // (zero == T_UNKNOWN)
    // handle the case where we don't know
```

in a GR algorithm.

One implication is that we must support *weakly normalized* representations of structures which in the exact case admit a strongly normalized (canonical) form. For example, a `gr_poly`

$$f = c_0 + c_1 X + \ldots + c_n X^n$$

is normalized by removing leading zero coefficients, but leading coefficients for which `gr_is_zero` returns *unknown* like $[0 \pm 0.001]$ or $0 + O(X^{10})$ will not be removed. Over a ring with inexactly represented elements, $\deg(f)$ will thus not necessarily be known exactly (but can be bounded). Operations that depend on knowing the exact degree (for example, polynomial division and GCD) will fail gracefully instead of computing bogus results.

This framework also accommodates ring implementations which principle have exact representation but in which some operations are not computable or decidable. An example is FLINT's `ca` type for exact real and complex numbers represented by means of algebraic or transcendental number fields $\mathbb{Q}(\alpha_1, \ldots, \alpha_n)$ which can have nontrivial (possibly undecidable) algebraic relations among the extension generators [20]. We deal correctly (e.g. by returning *unknown* in affected predicates) with situations where we cannot establish the necessary presence or absence of a relation.

GR also supports approximate implementations of rings without non-enclosure semantics, such "fields" of floating-point numbers, but does not treat them as genuine rings or fields (e.g. `gr_ctx_is_field` returns false).

8 Error handling

Since C lacks proper support for exceptions and workarounds emulating exceptions would be unsafe with FLINT's memory model, we implement error handling using return values and manual control flow. Most GR methods return a status code which is set to `GR_SUCCESS` (= 0) if an operation completes successfully. In the event of an error, one or both of the following flag bits will be set:

- A *domain error* (`GR_DOMAIN` flag) indicates that an operation cannot be assigned a mathematically meaningful result with the target type, for example:
  - Dividing by zero in a ring (with the possible exception of the zero ring, where one may unambiguously define $0/0 = 0$).
  - More generally, dividing elements which lack a quotient, e.g. $1/2$ in $\mathbb{Z}$.
  - Extracting the square root of a non-square element e.g. $\sqrt{-1}$ in $\mathbb{R}$.
  - Solving an inconsistent system of linear equations.
  - Evaluating a meromorphic function on $\mathbb{C}$ at a pole.
  - Accessing a vector out of bounds.
  - Multiplying matrices of incompatible shape.
- An *unable error* (`GR_UNABLE` flag) indicates that we cannot compute a result for implementation reasons, even if the result in principle may be mathematically well-defined, for example:
  - The result is too large to store in memory (e.g. expanding $(1 + X)^{10^{100}}$).
  - The result does not fit the finite representation of the target type (e.g. converting 2^{65} to an implementation of $\mathbb{Z}$ that uses 64-bit integers).
  - The used algorithm requires distinguishing between $x = 0$ and $x \neq 0$, and has no fallback strategy when the truth value cannot be decided.
  - The computation would be unreasonably long (e.g. factoring a 2048-bit integer).
  - No algorithm is implemented for this operation.

Status flags from consecutive operations can be bitwise OR:ed and passed forward for later error handling. We note that a pure `GR_DOMAIN` flag arising from a sequence of operations certainly indicates a domain error, while a combination of `GR_UNABLE` and `GR_DOMAIN` flags must be interpreted as an unable error.

Some GR algorithms handle and internally recover from (certain classes of) errors; in most cases, status codes will simply be propagated to the user. GR functions are marked with the `warn_unused_result` attribute allowing compilers like GCC to emit a warning if the user (or FLINT developer) accidentally discards a status code.

The distinction between domain and unable errors can be useful in high-level algorithms. For example, suppose that we try to compute A^{-1} given a matrix $A \in \mathbb{R}^{n \times n}$. We can try a combination of algorithms and representations (inexact, exact), starting with one optimized for speed. A domain error shows that A is not invertible in which case we can terminate; an unable error suggests that a slower but more powerful algorithm may succeed. This ability is closely related to the use of triple-valued logic for predicates discussed previously.

An important point is that we report domain errors instead of extending mathematically well-defined algebraic structures with inappropriate special values. While the FLINT type `arb` supports the special values $+\infty$, $-\infty$ and NaN ("not-a-number") via its non-generic interface, the GR implementation of $\mathbb{R}$ based on `arb` prevents creating those values (but does allow creating the interval $(0 \pm \infty)$ representing a completely unknown real number).

We could in principle avoid domain errors by assigning "junk values" to the undefined cases of partial functions. For example, we could define a total division on $\mathbb{Q}$ such that $1/0 = 0$. This is the standard approach taken in many proof assistants. Despite several potential benefits, we have not opted for such a solution since this gives up a useful ability to catch logical errors and bad user input. In any case, there would be little sense in returning junk values for unable errors; here we believe that the GR approach to error handling is reasonable.

Currently, GR lacks the ability to recover from out-of-memory conditions in builtin FLINT types such as `fmpz`, but this is a possible future improvement which in principle poses no problem for the GR programming model. Catching excessively large results is currently supported for the `qqbar` and `ca` types for exact algebraic, real and complex numbers: for example, one can prevent `qqbar` from generating algebraic numbers whose minimal polynomial over $\mathbb{Z}$ exceeds a prescribed degree or height bound. Such "evaluation bounds" are in many ways analogous to precision bounds for inexact representations and useful in hybrid algorithms where one ideally wants to be able to try several solution tactics with the ability to recover gracefully from a poorly-performing tactic rather than crashing, hanging, or returning bogus results

9 Correctness testing

We try to ensure correctness in GR through randomized testing: we generate many random elements and verify that supported operations satisfy all expected properties (commutativity, associativity, functional equations, etc). Ring implementations are expected to provide non-uniform random element generators for testing purposes to increase the chances of hitting corner cases. We note that generically written test routines in GR are compatible with virtually any particular ring implementation thanks to our unified error handling and support for inexact representation.

One of the clear drawbacks of pointer-based generic programming in C is that one gives up type safety and memory safety guarantees possible in languages with builtin generic support. In the author's experience, GR code does require a somewhat higher debugging effort than non-generic FLINT code, but the reliability of GR code with corresponding unit tests is probably on par with other parts of FLINT.

There is currently much interest in formally verified computer algebra, with several avenues of research:

- One possible approach is to implement a computer algebra system from scratch directly in a proof assistant like Lean or Coq. While the "calculational" abilities of proof assistants have improved remarkably in recent years (see [22, 23, 9] for some

examples) they remain well behind the capabilities traditional computer algebra systems and mathematical libraries.
- A different approach is to to formally verify existing libraries or use proof assistants to generate library code (e.g. in C) which is correct by construction [4, 25, 24].
- Yet another approach is to let "unsafe" systems like FLINT produce results which can be certified a posteriori in a formally verified way, e.g. for polynomial factorization [8].

The author has been asked on several occasions about the prospects of formally verifying parts of FLINT. Doing so from the bottom up seems extremely difficult due to the extensive entanglement between low-level optimizations and mathematical logic in the FLINT codebase. However, GR may help by creating two cleanly separated levels of abstraction. It should be feasible to develop a formal specification of the GR programming interface and formally verify some high-level generic GR algorithms, independently of which one could attempt to formally verify some specific ring implementations. Alternatively, we believe that ideas from FLINT and GR could be used in certified libraries developed from a clean slate.

Acknowledgements Fredrik Johansson has been supported by the ANR grant NuSCAP (ANR-20-CE-48-0014). We give special recognition to Nicolas Brisebarre (PI of NuSCAP) whose vision for a mathematical computation system with support for multiple levels of fidelity has helped motivate the development of GR. The author also thanks all FLINT contributors, with special thanks to Albin Ahlbäck for his help with assembly optimizations and general maintenance of the project.

References

1. John Abbott and Anna M Bigatti. CoCoALib: a C++ library for doing computations in commutative algebra, 2019.
2. Albin Ahlbäck and Fredrik Johansson. Fast basecases for arbitrary-size multiplication. working paper or preprint, January 2025.
3. Reimer Behrends, Kevin Hammond, Vladimir Janjic, Alexander Konovalov, Steve Linton, Hans-Wolfgang Loidl, Patrick Maier, and Phil Trinder. HPC-GAP: engineering a 21st-century high-performance computer algebra system. *Concurrency and Computation: Practice and Experience*, 28(13):3606–3636, January 2016.
4. Y. Bertot, N. Magaud, and P. Zimmermann. A proof of GMP square root. *Journal of Automated Reasoning*, 29(3-4):225–252, 2002.
5. Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B Shah. Julia: A fresh approach to numerical computing. *SIAM review*, 59(1):65–98, 2017.
6. Wieb Bosma, John Cannon, and Catherine Playoust. The Magma algebra system I: The user language. *Journal of Symbolic Computation*, 24(3-4):235–265, 1997.
7. Alexander Brandt, Robert H. C. Moir, and Marc Moreno Maza. *Employing C++ Templates in the Design of a Computer Algebra Library*, page 342–352. Springer International Publishing, 2020.
8. James H. Davenport. Towards verified polynomial factorisation. In *2024 26th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*, page 45–48. IEEE, September 2024.
9. James Harold Davenport. First steps towards computational polynomials in Lean, 2024.
10. Wolfram Decker, Christian Eder, Claus Fieker, Max Horn, and Michael Joswig, editors. *The Computer Algebra System OSCAR: Algorithms and Examples*, volume 32 of *Algorithms and Computation in Mathematics*. Springer, 1 edition, 2025.

11. Wolfram Decker, Gert-Martin Greuel, Gerhard Pfister, and Hans Schönemann. SINGULAR 4-4-0 — A computer algebra system for polynomial computations. `http://www.singular.uni-kl.de`, 2024.
12. Jean-Guillaume Dumas, Thierry Gautier, Mark Giesbrecht, Pascal Giorgi, Bradford Hovinen, Erich Kaltofen, B David Saunders, Will J Turner, Gilles Villard, et al. LinBox: A generic library for exact linear algebra. In *Proceedings of the 2002 International Congress of Mathematical Software, Beijing, China*, pages 40–50, 2002.
13. Claus Fieker, William Hart, Tommy Hofmann, and Fredrik Johansson. Nemo/Hecke: computer algebra and number theory packages for the Julia programming language. In *Proc. of the 42nd Intl. Symposium on Symbolic and Algebraic Computation*, ISSAC '17, pages 157–164. ACM, 2017.
14. FLINT developers. FLINT: Fast Library for Number Theory, 2025. Version 3.3.0, `http://flintlib.org`.
15. Laurent Fousse, Guillaume Hanrot, Vincent Lefèvre, Patrick Pélissier, and Paul Zimmermann. MPFR: A multiple-precision binary floating-point library with correct rounding. *ACM Transactions on Mathematical Software*, 33(2):13:1–13:15, June 2007.
16. GMP development team. GMP: The GNU Multiple Precision Arithmetic Library. `http://gmplib.org`, 2024.
17. William B. Hart. ANTIC: Algebraic number theory in C. *Computeralgebra-Rundbrief: Vol. 56*, 2015.
18. Fredrik Johansson. Arb: Efficient arbitrary-precision midpoint-radius interval arithmetic. *IEEE Transactions on Computers*, 66(8):1281–1292, August 2017.
19. Fredrik Johansson. Faster arbitrary-precision dot product and matrix multiplication. In *2019 IEEE 26th Symposium on Computer Arithmetic (ARITH)*. IEEE, June 2019.
20. Fredrik Johansson. Calcium: computing in exact real and complex fields. In *ISSAC '21*, 2021.
21. Xin Li, Marc Moreno Maza, and Éric Schost. On the virtues of generic programming for symbolic computation. In *Computational Science - ICCS 2007*, page 251–258. Springer Berlin Heidelberg, 2007.
22. Assia Mahboubi. *Machine-checked computer-aided mathematics*. Habilitation à diriger des recherches, Université de Nantes (UN), Nantes, FRA., January 2021.
23. Guillaume Melquiond. *Formal Verification for Numerical Computations, and the Other Way Around*. Habilitation à diriger des recherches, Université Paris Sud, Orsay, France, 2019.
24. Guillaume Melquiond and Josué Moreau. A safe low-level language for computer algebra and its formally verified compiler. *Proceedings of the ACM on Programming Languages*, 8(ICFP):121–146, August 2024.
25. Guillaume Melquiond and Raphaël Rieu-Helft. A Why3 framework for reflection proofs and its application to GMP's algorithms. In *Automated Reasoning: 9th International Joint Conference, IJCAR 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings 9*, pages 178–193. Springer, 2018.
26. Aaron Meurer et al. SymPy: symbolic computing in Python. *PeerJ Computer Science*, 3:e103, January 2017.
27. The PARI Group, University of Bordeaux. *PARI/GP version 2.17.2*, 2025. `http://pari.math.u-bordeaux.fr/`.
28. The Sage Developers. *SageMath, the Sage Mathematics Software System (Version 10.6)*, 2025. `https://www.sagemath.org`.
29. J. van der Hoeven, G. Lecerf, B. Mourrain, P. Trébuchet, J. Berthomieu, D. N. Diatta, and A. Mantzaflaris. Mathemagix: the quest of modularity and efficiency for symbolic and certified numeric computation? *ACM Communications in Computer Algebra*, 45(3/4):186–188, January 2012. `http://mathemagix.org`.
30. A. Waksman. On Winograd's algorithm for inner products. *IEEE Transactions on Computers*, C–19(4):360–361, April 1970.