On the Virtues of Generic Programming for Symbolic Computation

Xin Li, Marc Moreno Maza and Éric Schost
{xli96, moreno, schost}@scl.csd.uwo.ca

University of Western Ontario, London N6A 1M8

Abstract. The purpose of this study is to measure the impact of C level code polynomial arithmetic on the performances of AXIOM high-level algorithms, such as polynomial factorization. More precisely, given a high-level AXIOM package `P` parametrized by a univariate polynomial domain `U`, we have compared the performances of `P` when applied to different `U`'s, including an AXIOM wrapper for our C level code.
Our experiments show that when `P` relies on `U` for its univariate polynomial computations, our specialized C level code can provide a significant speed-up. For instance, the improved implementation of square-free factorization in AXIOM is 7 times faster than the one in MAPLE and very close to the one in MAGMA. On the contrary, when `P` does not rely much on the operations of `U` and implements its private univariate polynomial operation, then `P` cannot benefit from our highly optimized C level code. Consequently, code which is poorly generic reduces the speed-up opportunities when applied to highly efficient and specialized polynomial data-types.

Keywords: Generic Programming, fast arithmetic, high performance, polynomials.

1 Introduction

Generic programming, and in particular type constructors parametrized by types and values, is a clear need for implementing computer algebra algorithms. This has been one of the main motivations in the development of computer algebra systems and languages such as AXIOM [10] and ALDOR [15] since the 1970's. AXIOM and ALDOR have a two-level object model of *categories* and *domains* which allows the implementation of algebraic structures (rings, fields, ...) and their members (polynomial domains, fields of rational functions, ...). In these languages, the user can implement domain and category constructors, that is, functions returning categories or domains. For instance, one can implement a function `UP` taking a ring `R` as parameter and returning the ring of univariate polynomials over `R`. This feature is known as *categorical programming*.

Another goal in implementing computer algebra algorithms is that of efficiency. More precisely, it is desirable to be able to realize successful implementations of the best algorithms for a given problem. Sometimes, this may sound contradictory with the generic programming paradigm. Indeed, efficient implementations often require specialized data-structures (e.g., primitive arrays of machine words for encoding dense univariate polynomials over a finite field). High-performance was not the primary concern in the development AXIOM. For instance, until recently [11], AXIOM had no domain constructor for univariate polynomials with dense representation.

The MAGMA [2, 1] computer algebra system, developed at the University of Sydney since the 1990's, has succeeded in providing both generic types and high-performance. As opposed to many previous systems, a strong emphasis was put on performance: asymptotically fast state-of-the art algorithms are implemented in MAGMA, which has become a *De facto* reference regarding performance.

MAGMA's design uses the language of universal algebra as well. Users can dynamically define and compute with structures (groups, rings, ...), that belong to categories (e.g., permutation groups), which themselves belong to varieties (e.g., the variety of groups); these algebraic structures are first-class objects. However, some aspects of categorical programming available in AXIOM are not present: users cannot define new categories; the interfacing with C seems not possible either.

In this paper, we show that generic programming can contribute to high-performance. To do so, we first observe that dense univariate and multivariate polynomials over finite fields play a central role in computer algebra, thanks to modular algorithms. Therefore, we have realized highly optimized implementations of these polynomial data-types in C, ALDOR and LISP. This work is reported in [5] and [12].

The purpose of this new study is to measure the impact of our C level code polynomial arithmetic on the performances of AXIOM high-level algorithms, such as factorization. More precisely, given a high-level AXIOM package `P` (or domain) parametrized by a univariate polynomial domain `U`, we have compared the performances of `P` when applied to different `U`'s, including an AXIOM wrapper for our C level code.

Our experiments show that when `P` relies on `U` for its univariate polynomial computations, our specialized C level code can provide a significant speed-up. On the contrary, when `P` does not rely much on the operations of `U` and implements its private univariate polynomial operations, then `P` cannot benefit from our highly optimized C level code. Consequently, code which is poorly generic reduces the speed-up opportunities when applied to highly efficient and specialized polynomial data-types.

2 Software Overview

We present briefly the AXIOM polynomial domain constructors involved in our experimentation. Then, we describe the features of our C code that play a central in this study: finite field arithmetic and fast univariate polynomial arithmetic. We notably discuss how the choice of *special primes* enables us to obtain fast algorithms for reduction modulo p.

2.1 AXIOM **polynomial domain constructors**

Let `R` be an AXIOM `Ring`. The domain `SUP(R)` implements the ring of univariate polynomials with coefficients in `R`. The data representation of `SUP(R)` is *sparse*, that is, only non-zero terms are encoded. The domain constructor `SUP` is written in the AXIOM language.

The domains `DUP(R)` implements exactly the same operations as `SUP(R)`. More precisely, these two domains satisfy the category `UnivariatePolynomial-Category(R)`. However, the representation of the latter domain is dense: all terms, null or not, are encoded. The domain constructor `DUP` was developed in the AXIOM language, see [11] for details.

Another important domain constructor in our study is `PF`: for a prime number `p`, the domain `PF(p)` implements the prime field $\mathbb{Z}/p\mathbb{Z}$.

Our C code is dedicated to multivariate polynomials with dense representation and coefficients in a prime field To make this code available at the AXIOM level, we have implemented a domain constructor `DUP2` wrapping our C code. For a prime number `p`, the domains `DUP2(p)` and `DUP(PF(p))` implement the same category, that is, `UnivariatePolynomialCategory(PF(p))`.

2.2 Finite field arithmetic

The implementation reported here focuses on some *special* small finite fields. By a *small* finite field, we mean a field of the form $\mathbb{K} = \mathbb{Z}/p\mathbb{Z}$, for p a prime that fits in a 26 bit word (so that the product of two elements reduced modulo p fits into a `double` register). Furthermore, the primes p we consider have the form $k2^\ell + 1$, with k a *small* odd integer (typically $k \leq 7$), which enables us to write specific code for Euclidean division.

The elements of $\mathbb{Z}/p\mathbb{Z}$ are represented by integers from 0 to $p-1$. Additions and subtractions in $\mathbb{Z}/p\mathbb{Z}$ are performed in a straightforward way: we perform integer operations, and the result is then reduced modulo p. Since the result of additions and subtractions is always in $-(p-1), \ldots, 2(p-1)$, modular reduction requires at most a single addition or subtraction of p; for the reduction, we use routines coming from Shoup's NTL library [9, 14].

Multiplication in $\mathbb{Z}/p\mathbb{Z}$ requires more work. A standard solution, present in NTL, consists in performing the multiplication in `double` precision floating-point registers, compute numerically the quotient appearing in the Euclidean division by p, and finally deduce the remainder.

Using the special form of the prime p, we designed the following faster "approximate" Euclidean division, that shares similarities with Montgomery's REDC algorithm [13]; for another use of arithmetic modulo special primes, see [4]. Let thus Z be in $0, \ldots, (p-1)^2$; in actual computations, Z is obtained as the product of two integers less than p. The following algorithm computes an approximation of the remainder of kZ by p, where we recall that p has the form $k2^\ell + 1$:

1. Compute $q = \lfloor \frac{Z}{2^\ell} \rfloor$.
2. Compute $r = k(Z - q2^\ell) - q$.

Proposition 1. *Let r be as above and let $r_0 < p$ be the remainder of kZ by p. Then $r \equiv r_0 \bmod p$ and $r = r_0 - \delta p$, with $0 \le \delta < k+1$.*

Proof. Let us write the Euclidean division of kZ by p as $kZ = q_0 p + r_0$. This implies that

$$q = q_0 + \left\lfloor \frac{q_0 + r_0}{k2^\ell} \right\rfloor$$

holds. From the equality $qp + r = q_0 p + r_0$, we deduce that we have

$$r = r_0 - \delta p \quad \text{with} \quad \delta = \left\lfloor \frac{q_0 + r_0}{k2^\ell} \right\rfloor p.$$

The assumption $Z \le (p-1)^2$ enables us to conclude that $\delta < k+1$ holds. □

In terms of operations, this reduction is faster than the usual algorithms, which rely on either Montgomery's REDC or Shoup's floating-point techniques. The computation of q is done by a logical shift; that of r requires a logical AND (to obtain $Z - 2^\ell q$), and a single multiplication by the constant c. Classical reduction algorithms involve 2 multiplications, and other operations (additions and logical operations). Accordingly, in practical terms, our approach turned out to be more efficient.

There are however drawbacks to this approach. First, the algorithm above does not compute $Z \bmod p$, but a number congruent to kZ modulo p (this multiplication by a constant is also present in Montgomery's approach). This is however easy to circumvent in several cases, for instance when doing multiplications by precomputed constants (this is the case in FFT polynomial multiplication, see below), since a correcting factor $k^{-1} \bmod p$ can be incorporated in these constants. The second drawback is that the output of our reduction routine is not reduced modulo p. When results are reused in several computations, errors accumulate, so it is necessary to perform some error reduction at regular time steps, which slows down the computations.

2.3 Polynomial arithmetic

For polynomial multiplication, we use the Fast Fourier Transform (FFT) [6, Chapter 8], and its variant the Truncated Fourier Transform [8]. Indeed, since we work modulo primes p of the form $k2^\ell + 1$, Lemma 8.8 in [6] shows that $\mathbb{Z}/p\mathbb{Z}$ admits 2^ℓth primitive roots of unity, so that it is suitable for FFT multiplication for output degrees up to $2^\ell - 1$.

Both variants feature a $O(d \log(d))$ asymptotic complexity; the latter offers a smoother running time, avoiding the usual abrupt jumps that occur at powers of 2 in classical Fast Fourier Transforms.

Using fast multiplication enables us to write a fast Euclidean division for polynomials, using Cook-Sieveking-Kung's approach through power series inversion [6, Chapter 9]. Recall that this algorithm is based on the remark that the quotient q in the Euclidean division $u = qv + r$ in $\mathbb{K}[x]$ satisfies

$$\mathrm{rev}_{\deg u - \deg v}(q) = \mathrm{rev}_{\deg u}(u)\, \mathrm{rev}_{\deg v}(v)^{-1} \bmod x^{\deg u - \deg v + 1},$$

where $\text{rev}_m(p)$ denotes the *reverse* polynomial $x^m p(1/x)$. Hence, computing the quotient q is reduced to a power series division, which itself can be done in time $O(d \log(d))$ using Newton's iteration [6, Chapter 9].

Newton's iteration was implemented using *middle product techniques* [7], which enable us to reduce the cost of a direct implementation by a constant factor (these techniques are particularly easy to implement when using FFT multiplication, and are already described in this case in [14].

Our last ingredient is GCD computation. We implemented both the classical Euclidean algorithm, as well as its faster divide-and-conquer variant using so-called Half-GCD techniques [6, Chapter 11]. The former features a complexity in $O(d^2)$, whereas the latter has cost in $O(d \log(d)^2)$, but is hindered by a large multiplicative constant hidden by the big-O notation.

2.4 Code connection

Open AXIOM is based on GNU Common Lisp (GCL). And GCL is developed in C [12]. We follow the GCL developers' approach to integrate our C level code into GCL's kernel. The crucial step is converting different polynomial data representations between AXIOM and the ones in our C library via GCL level. The overhead of these conversions may significantly reduce the effectiveness of our C implementation. Thus, good understanding of data structures in AXIOM and GCL is a necessity to establish an efficient code connection.

3 Experimentation

In this section, we compare our specialized domain constructor `DUP2` with our generic domain constructor `DUP` and the standard AXIOM domain constructor `SUP`. Our experimentation takes place into the polynomial rings:

- $A_p = \mathbb{Z}/p\mathbb{Z}[x]$,
- $B_p = (\mathbb{Z}/p\mathbb{Z}[x]/\langle m \rangle)[y]$,

for a machine word prime number p and an irreducible polynomial $m \in \mathbb{Z}/p\mathbb{Z}[x]$. The ring A_p can be implemented by any of the three domain constructors `DUP2`, `DUP` and `SUP` applied to `PF(p)`, whereas B_p is implemented by either `DUP` and `SUP` applied to A_p. In both A_p and B_p, we compare the performances of factorization and resultant computations provided by theses different constructions. These experimentations deserve two goals.

(G_1) When there is a large proportion of the running time which is spent in computing products, remainders, quotients, GCDs in A_p, we believe that there are opportunities for significant speed-up when using `DUP2` and we want to measure this speed-up w.r.t. `SUP` and `DUP`.

(G_2) When there is a little proportion of the running time which is spent in computing products, remainders, quotients, GCDs in A_p, we want to check whether using `DUP2`, rather than `SUP` and `DUP`, could slow down computations.

For computing univariate polynomial resultants over a field, AXIOM runs the package `PseudoRemainderSequence` implementing the algorithms of Ducos [3]. This package takes `R: IntegralDomain` and `polR: UnivariatePolynomialCategory(R)` as parameters. However, this code has its private `divide` operation and does not rely on the one provided by the domain `polR`. In fact, the only non-trivial operation that will be run from `polR` is addition! Therefore, if `polR` has a fast division with remainder, this will not benefit to resultant computations performed by the package `PseudoRemainderSequence`. Hence, in this case, there is very little opportunities for `DUP2` to provide speed-up w.r.t. `SUP` and `DUP`.

For square-free factorization over a finite field, AXIOM runs the package `UnivariatePolynomialSquareFree`. It takes `RC: IntegralDomain` `P: UnivariatePolynomialCategory(RC)` as parameters. In this case, the code relies on the operations `gcd` and `exquo` provided by `P`. Hence, if `P` provided fast gcd computations and fast divisions, this will benefit to the package `UnivariatePolynomialSquareFree`. Hence, in this case, there is a potential for `DUP2` to speed-up computations w.r.t. `SUP` and `DUP`.

We start the description of our experimental results with resultant computations in $A_p = \mathbb{Z}/p\mathbb{Z}[x]$. As mentioned above, this is not a good place for obtaining significant performance gain. Figure 1 shows that computations with `DUP2` are just slightly faster than those with `SUP`. In fact, we are happy to verify that using `DUP2`, which implies data-type conversions between the AXIOM and C data-structures, does not slow down computations!

We continue with square-free factorization and irreducible factorization in A_p. Figure 2 (resp. Figure 3) shows that `DUP2` provides a speed-up ratio of 8 (resp. 7) for polynomial with degrees about 9000 (resp. 400). This is due to the combination of the fast arithmetic (FFT-based multiplication, Fast division, Half-GCD) and highly optimized code of this domain constructor.

In the case of irreducible factorization, we could have obtained a better ratio if the code was *more generic*. Indeed, the irreducible factorization over finite fields in AXIOM involves a package which has its private univariate polynomial arithmetic, leading to a problem similar to that observed with resultant computations. The package in question is `ModMonic`, parametrized by `R: Ring` and `Rep: UnivariatePolynomialCategory(R)`, which implements the Frobenius map.

We conclude this section with our benchmarks in $B_p = (\mathbb{Z}/p\mathbb{Z}[x]/\langle m \rangle)[y]$. For resultant computations in B_p the speed-up ratio obtained with `DUP2` is better than in the case of A_p. This is because the arithmetic operations of `DUP2` (addition, multiplication, inversion) perform better than those of `SUP` or `DUP`. Finally, for irreducible factorization in B_p, the results are quite surprising. Indeed, AXIOM uses Trager's algorithm (which reduces computations to resultants in B_p, irreducible factorization in A_p and gcds in B_p) and, based on our previous results, we could have anticipated a good speed-up ratio. Unfortunately, the package `AlgFactor`, which is used for algebraic factorization, has its private arithmetic. More precisely, it "re-defines" B_p with `SUP` and factorizes the input polynomial over this new B_p.

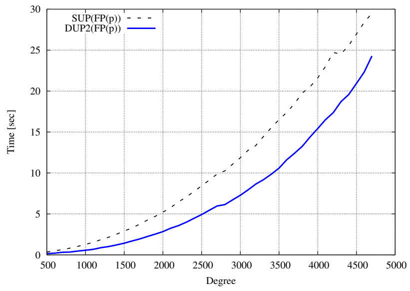

Fig. 1. Resultant computation in $\mathbb{Z}/p\mathbb{Z}[x]$

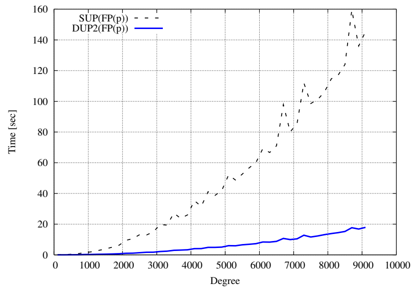

Fig. 2. Square-free factorization in $\mathbb{Z}/p\mathbb{Z}[x]$

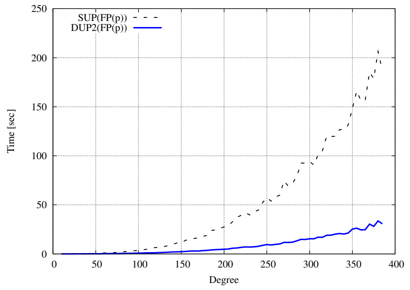

Fig. 3. Irreducible factorization in $\mathbb{Z}/p\mathbb{Z}[x]$

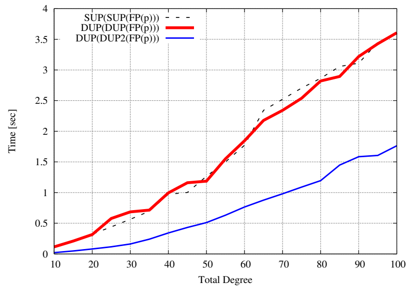

Fig. 4. Resultant computation in $(\mathbb{Z}/p\mathbb{Z}[x]/\langle m \rangle)[y]$

4 Conclusion

The purpose of this study is to measure the impact of our C level specialized implementation for fast polynomial arithmetic on the performances of AXIOM high-level algorithms. Generic programming is well designed in the AXIOM system. The experiment results demonstrate that by replacing a few important operations in `DUP(PF(p))` with our C level implementation, the original AXIOM univariate polynomial arithmetic over $\mathbb{Z}/p\mathbb{Z}$ has been speed up by a large factor in general. For algorithm such as univariate polynomial square free factorization over $\mathbb{Z}/p\mathbb{Z}$, the improved AXIOM code is 7 times faster than the one in MAPLE and very close to the one in MAGMA (see Figure 6).

References

1. W. Bosma, J. J. Cannon, and G. Matthews. Programming with algebraic structures: design of the Magma language. In *ISSAC'94*, pages 52–57. ACM Press, 1994.
2. The Computational Algebra Group in the School of Mathematics and Statistics at the University of Sydney. *The MAGMA Computational Algebra System for Algebra, Number Theory and Geometry*. `http://magma.maths.usyd.edu.au/magma/`.

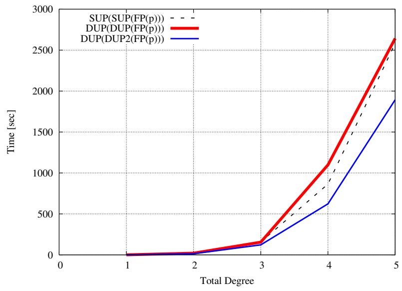

Fig. 5. Irreducible factorization in $(\mathbb{Z}/p\mathbb{Z}[x]/\langle m\rangle)[y]$

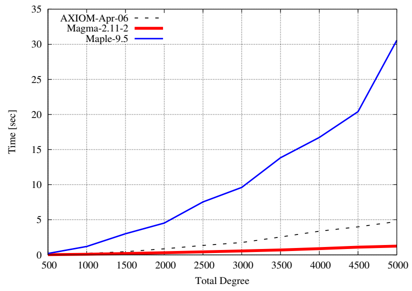

Fig. 6. Square-free factorization in $\mathbb{Z}/p\mathbb{Z}[x]$

3. L. Ducos. Optimizations of the subresultant algorithm. *J. of Pure Appl. Alg.*, 145:149–163, 2000.
4. T. Färnqvist. Number theory meets cache locality: efficient implementation of a small prime FFT for the GNU Multiple Precision arithmetic library. Master's thesis, Stockholms Universitet, 2005.
5. A. Filatei, X. Li, M. Moreno Maza, and É. Schost. Implementation techniques for fast polynomial arithmetic in a high-level programming environment. In *ISSAC'06*, pages 93–100. ACM Press, 2006.
6. J. von zur Gathen and J. Gerhard. *Modern Computer Algebra*. Cambridge University Press, 1999.
7. G. Hanrot, M. Quercia, and P. Zimmermann. The middle product algorithm, I. *Appl. Algebra Engrg. Comm. Comput.*, 14(6):415–438, 2004.
8. J. van der Hoeven. The truncated Fourier transform and applications. In *ISSAC'04*, pages 290–296. ACM Press, 2004.
9. http://www.shoup.net/ntl. *The Number Theory Library*. V. Shoup, 1996–2006.
10. R. D. Jenks and R. S. Sutor. *AXIOM, The Scientific Computation System*. Springer-Verlag, 1992.
11. X. Li. Efficient management of symbolic computations with polynomials, 2005. University of Western Ontario.
12. X. Li and M. Moreno Maza. Efficient implementation of polynomial arithmetic in a multiple-level programming environment. In A. Iglesias and N. Takayama, editors, *ICMS 2006*, pages 12–23. Springer, 2006.
13. P. L. Montgomery. Modular multiplication without trial division. *Math. of Comp.*, 44(170):519–521, 1985.
14. V. Shoup. A new polynomial factorization algorithm and its implementation. *J. Symb. Comp.*, 20(4):363–397, 1995.
15. S. M. Watt, P. A. Broadbery, S. S. Dooley, P. Iglio, S. C. Morrison, J. M. Steinbach, and R. S. Sutor. A first report on the A# compiler. In *ISSAC'94*, pages 25–31. ACM Press, 1994.