Efficient Management of Symbolic Computation with Polynomials

by

Xin Li

Department of Computer Science

Master of Science

Faculty of Graduate Studies
The University of Western Ontario
London, Ontario
August 2005

ABSTRACT

Symbolic polynomial computation is widely used to solve many applied or abstract mathematical problems. Some of them, such as solving systems of polynomial equations, have exponential complexity. Their implementation is, therefore, a challenging task.

By using adapted data structures, asymptotically fast algorithms and effective code optimization techniques, we show how to reduce the practical and theoretical complexity of these computations. Our effort is divided into three categories: integrating the best known techniques into our implementation, investigating new directions, and measuring the interactions between numerous techniques.

We chose AXIOM and ALDOR as our implementation and experimentation environment, since they are both strongly typed and highly efficient Computer Algebra Systems (CAS). Our implementation results show that our methods have great potential to improve the efficiency of exact polynomial computations with the selected CASs. The performance of our implementation is comparable to that of (often outperforming) the best available packages for polynomial computations.

Keywords: Exact Symbolic Computation, Computer Algebra System, Polynomial Computations, Axiom, Aldor.

I wish to extend my appreciation and gratitude to my supervisor Dr. Marc Moreno Maza for his guidance, support and encouragement through this entire investigation.
I wish to thank Dr. Stephen Watt for his advice and the opportunity he gave to me to work in the Ontario Research Center for Computer Algebra.
I would also like to express my thanks to professor Éric. Schost for his great help.

Contents

List of Figures

Chapter 1

Introduction

1.1 What is the research about

Symbolic computation is a process of mathematical transformation for symbolic expressions which are designed to be implemented on computer systems. Our research area is symbolic computation with polynomials: the automatic transformation of polynomial systems in an exact way. This is not only an issue of exploring abstract mathematical theories, but also a subject of investigating methods for high performance polynomial computations. Our research work is emphasized on analyzing the practical and theoretical complexity of these computations, and improving their performance in a significant manner. Our research work can be divided into three categories.

1. First, we have studied the best known techniques for symbolic polynomial computations and integrated them in our implementation environment. This is a process of analysis and synthesis. It requires a great amount of time to implement, compare and improve existing algorithms for symbolic polynomial manipulation, then measure the performance improvement, or even degradation, after plugging them into our implementation.

2. Second, we have investigated two new directions: exact polynomial computation modulo big primes and Fast Fourier Transform based multivariate mul-

tiplication, where no work or little work has been done by others. This is a process to verify new ideas for efficient implementation. It demands rigorous proofs and appropriate benchmarks.

3. Third, we also aim at measuring precisely the effectiveness of the techniques we used and their interactions. Our research produced many interesting results and we will make detailed comments about this in the following chapters.

1.2 Motivations

The desire of achieving high performance, and supporting new methods for symbolic polynomial computations, such as multivariate Hensel lifting for solving systems of equations, are the two main motivations for this study.

As we mentioned above, exact symbolic computation with polynomials usually consumes a large amount of memory and CPU time. When the size of the problem is large, which is the usual case, the challenges are

- overcoming computational bottlenecks such as intermediate expression swell and,
- producing the results within an acceptable length of time.

In fact, expression swell [18] (that is the fact that the size of expressions grows dramatically during a calculation) is a well known problem in symbolic computation with polynomials. If this happens during intermediate stages, we call it intermediate expression swell. Why does this happen? The reason is simple: symbolic computations manipulate numbers by using their mathematical definitions rather than using floating point approximations. Hence, they need to keep exact results at each step. Consequently, during the middle stages of a calculation, intermediate expressions can expand substantially, even if the final result is comparatively small. Computations like polynomial factorization and polynomial greatest common divisor are good examples. We will explain this in Chapter 3, Section 1. Obviously,

when the size of an expression grows, we need more memory to store it and longer CPU time to transform it. When the growth becomes exponential, the computer system will slow down dramatically and the computation is likely not to terminate. Therefore, computer algebra systems need to control this situation, and this is generally achieved through modular methods. We will give the definition of modular methods in the next section. However, in practice for these modular methods to give their best performance, one needs to use fast polynomial arithmetic as well as an efficient interface with the machine arithmetic. This is clearly a more difficult task in an interactive environment.

In addition, most comprehensive computer algebra systems, such as MAPLE, MAGMA, ALDOR, AXIOM are garbage collected systems. Those systems free programmers from the tedious tasks of memory management and leave garbage collectors to deal with it. But, when developers intend to write a high performance package or library, they have to carefully study the general mechanism of garbage collection, and create sample programs to test in which way the applications could better use memory through the help of garbage collectors. Hence, high performance exact symbolic computation with polynomials is a sophisticated topic. It requires developers to study fast polynomial arithmetic, to be experts on their implementation environment, including compiler's optimization and low level support, such as machine arithmetic and garbage collection.

Besides improving performance for general polynomial computations, we are also interested in Hensel lifting based modular methods (see Figure 1.1) and fast arithmetic modulo a polynomial ideal [18]. In these areas, there is still not a satisfactory solution toward high performance issues. This thesis provides preliminary results on these questions. We plan to continue investigating them in the near future.

1.3 Related work

A computer algebra system (CAS) by definition is a software program that facilitates symbolic mathematics.

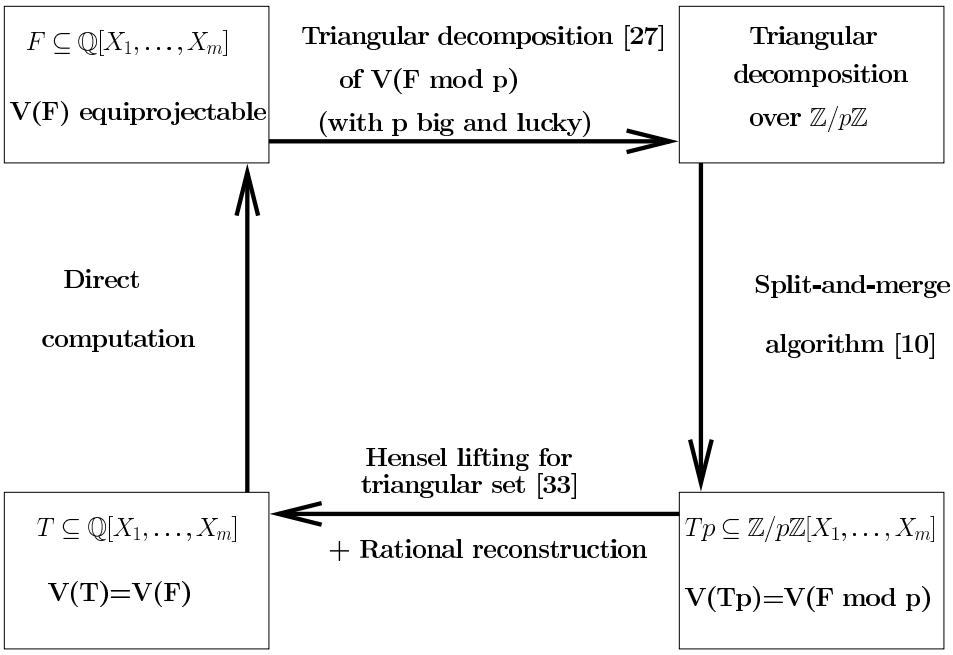

Figure 1.1: Hensel lifting technique for triangular decompositions.

General purpose CAS such as Derive, Macsyma, Maple, Mathematica, AXIOM, Aldor and Reduce intend to provide comprehensive functionalities which can facilitate mathematicians, scientists, and engineers to conduct their mathematical research work. There are also some systems which focus on specific topics, such as NTL [34], COCOA [5], MAGMA [7], GMP [16] and PARI/GP [6].

Since symbolic computation with polynomials has wide usage, most CAS provide a symbolic solver for solving systems of equations and other polynomial problems symbolically. However, not all of them are efficient when dealing with large scale polynomial computations. Moreover, even those which provide high performance, still require a continuous effort for improvement. Indeed, computer technology, as well as mathematical strategies, evolve quickly. A problem which was unsolved yesterday may become an easy task today.

Similarly, an algorithm which seemed unpractical yesterday may become a standard routine today. For instance, asymptotically fast methods for exact computations have been known for a quarter of a century. Unfortunately their impact on computer algebra systems has been reduced since it was believed that they were

irrelevant in practice. Let us quote [18] at p. 132: "In practice, depending on the method of recursion used, the fast method (the univariate multiplication based on the Fast Fourier Transform) is better than the classical method approximatively for $n+m \geq 600$ where n and m are the degree of the input polynomials." This *myth* died in the recent years and these methods permitted to increase the magnitude of effective computations in several areas like exact factorization of polynomials. For instance, for the fast algorithm (FFT-based univariate multiplication) mentioned above, the cross-over point is today $n+m \geq 64$ in [35] p.363-397.

1.4 Objectives and results

As mentioned above, our research work is emphasized on the following aspects. First, we integrate together known techniques for improving symbolic computations with polynomials. We summarize them below.

Modular methods. One of the main successes of the computer algebra community in the last 30 years is the discovery of algorithms, called modular methods, that allow to keep the swell of intermediate expressions under control. Mathematical high-level algorithms with polynomials are generally based on modular methods through which the size of a problem is reduced. Our implementation results in Chapter 3 illustrate that the overall performance of the studied problem (computing polynomial GCDs over algebraic number fields) is improved in a significant manner by using modular methods.

Adapted Data Representation. Selecting adapted data structures to represent mathematical objects in memory is crucially important for high performance. Both time and space complexity have to be taken into account. Moreover, a given family of mathematical objects, say polynomials or matrices, may require different representations for different calculations. Therefore, we need to choose adapted data representations for each application and develop efficient transformation mechanisms between those different representations; ideally, such a transformation must run in

linear time. Our benchmarks of Chapters 3, 4 and 7 illustrate the performance improvement by using adapted data representations compared to generic data representations.

Machine-level optimized implementation. Many efficiency-critical operations in our implementation are programmed in assembly language and C language. The purpose is to effectively use machine level arithmetic and the computer architecture. Certainly, compilers intend to do this for us, but they do not always make use of the latest features of some computer architectures. Moreover, programmers have a clear view of their tasks in advance, they should either tune their programs for the compiler to better optimize them, or explicitly take advantage of a specific computer-architecture. This is not an easy task, as it requires a good knowledge of low-level languages and hardware. Our work in this area is reported in Chapter 5.

Fast polynomial arithmetic. We implement fast univariate polynomial multiplication over finite fields based on the Discrete Fourier Transform. Theoretically, this algorithm has lower computational complexity than any other known method.

However, obtaining an efficient implementation is challenging [13, 17]. We put great effort on this algorithm since it is the key for obtaining fast algorithms for many other polynomial operations, such as division, GCD and others [13]. Our results are satisfactory.

- Our implementation of the FFT-based univariate polynomial multiplication over $\mathbb{Z}/p\mathbb{Z}$ (see Chapter 6) outperforms the most efficient package currently available, namely NTL [34] when p is a small prime.

- Our implementation of the FFT-based multivariate polynomial multiplication (see Chapter 7) outperforms the package which, to our knowledge, is the best one for this task, namely MAGMA.

Based on this work, we plan to develop highly efficient implementation of polynomial division and GCD in the near future.

Garbage Collection. Currently, most popular Garbage Collection Systems in CAS are tracing GC. Typically, tracing GC collects objects that are no longer reachable along a path of references starting from a set of root references. As mentioned above, for a high-performance application which runs in a garbage collected environment, the mechanism and effectiveness of the garbage collector need to be considered. Data structures which are easy to recycle and algorithms which are garbage collection friendly can enhance the performance in a significant manner. We have studied the garbage collection system of ALDOR and implemented a package for the ALDOR compiler. This package collects liveness information of variables and helps the ALDOR garbage collector reclaim unreachable objects earlier than only using the information of reachability. By using our package at compile-time, the memory consumption of many polynomial computations will be reduced at run-time. This implementation work is presented in Chapter 8.

In this thesis, we have also investigated new strategies for polynomial computations. We summarize them below.

Developing specific techniques for computations modulo big primes. Modular computations are well developed for primes that fit in one machine-word [17]. However, certain classes of algorithms, such as Hensel lifting for solving non-linear multivariate polynomial systems [10] require the use of big primes. Those algorithms work on a prime field $\mathbb{Z}/p\mathbb{Z}$ for a large prime number p:

- either directly by computing with big integers and then reducing modulo p,
- or by computing modulo several small primes and then recombining the results with the Chinese Remaindering Algorithm.

We study these two approaches in Chapter 6.

Implementing FFT-based arithmetic for multivariate polynomials. FFT-based techniques are well developed for univariate polynomials [17]. However, very few solutions have emerged yet for the multivariate case [13, 31]. We implemented

FFT-based multivariate multiplication via the Kronecker substitution. We also provide a theoretical study of this strategy. Our results are reported in Chapter 7. Our implementation of this strategy outperforms the package which, to our knowledge, is the best one for this task, namely MAGMA.

Chapter 2

Implementation environment

2.1 History

AXIOM [22] is a comprehensive Computer Algebra System which has been in development since 1971. It was originally developed by IBM under the direction of Richard Jenks. At that time AXIOM was called Scratchpad. In the 1990s, it was renamed AXIOM and sold as a commercial software by the Numerical Algorithms Group (NAG). In 2002, it became a free and open source software, extended and maintained by a team under the lead of Timothy P. Daly.

On top of its many UNIX versions, the AXIOM system also has a Windows version. The user can download the AXIOM source tree or the pre-compiled binaries from `http://page.axiom-developer.org/`. It includes integrated graphics and a document browser as shown on Figure 2.1 and Figure 2.2.

We chose AXIOM as our implementation and experimentation environment based on four reasons:

1. AXIOM provides both an interactive interpreter and a high-level language compiler.

2. AXIOM, with its different implementation levels (see Figure 2.5 p. 16), provides a unique development environment among all CAS.

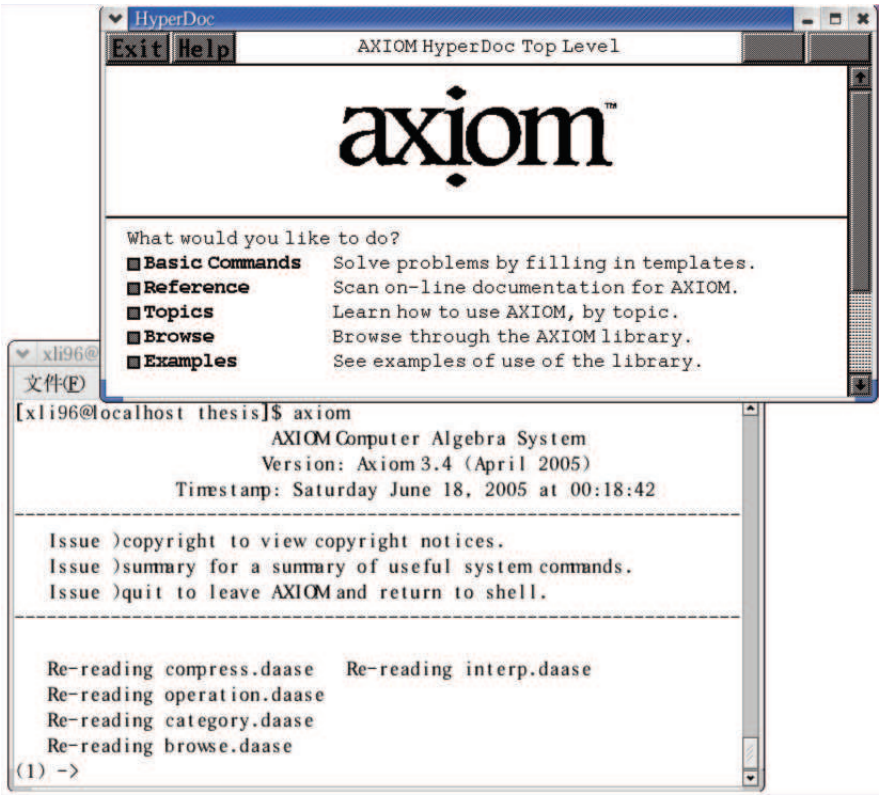

Figure 2.1: AXIOM interactive mode.

3. As shown in this thesis, AXIOM is well adapted for high performance computer algebra.

4. AXIOM is free and open source.

AXIOM has a very high level programming language called SPAD, the abbreviation of Scratchpad. It can be compiled into COMMON LISP by its own built-in compiler. There is an external stand-alone compiler implemented in C which also accepts the SPAD language, called ALDOR [2, 38]. ALDOR was originally developed by a team under the direction of Stephen Watt at IBM T.J. Watson Research Center. The ALDOR compiler has been available as part of the AXIOM system since late 1994, and is now released as an independent software.

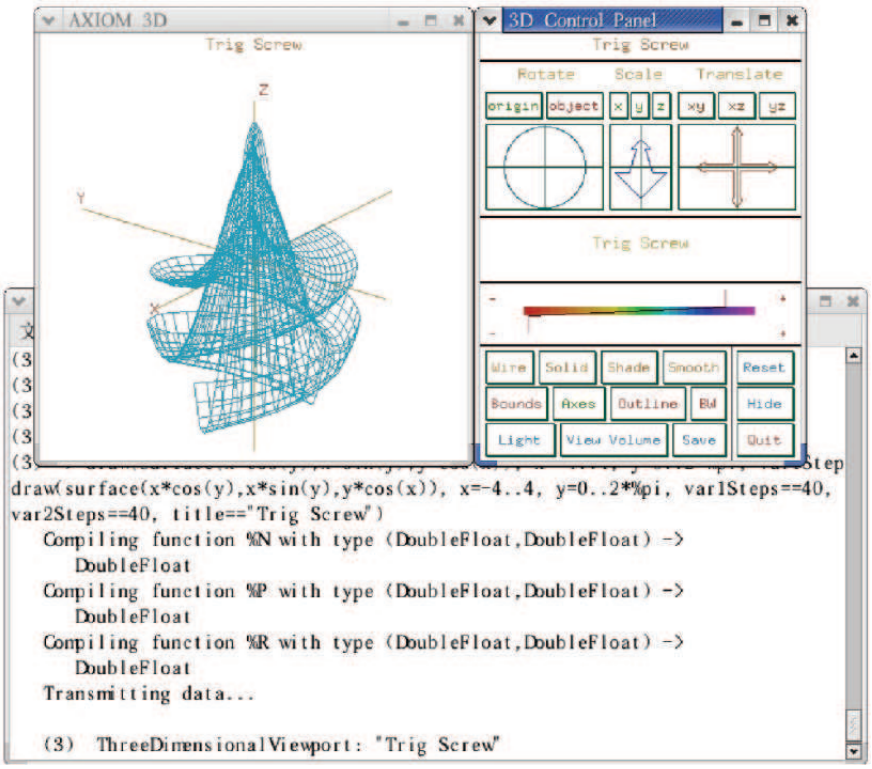

Figure 2.2: AXIOM graphics.

2.2 Overview

AXIOM has both an interactive mode for user interactions and a programming language for building library modules. In the interactive mode, users can evaluate arithmetic expressions, declare and define variables, call library functions and define their own functions. Figure 2.3 shows how to create a Hilbert Matrix in AXIOM's interactive mode. The (i,j)-th entry of a Hilbert matrix is given by $1/(i+j+1)$.

The user can also collect AXIOM statements and commands into files, called *"input" files*, and then run them as scripts in the interactive mode. Actually, any thing that one can enter directly in the AXIOM interactive mode can be put into an input file. This provides programmers with a very convenient way to write functions and large blocks of programs, then test and run them.

AXIOM has an efficient mechanism to compile and execute programs. When AXIOM can completely determine the type of every object in a program, it trans-

```
→ a:Matrix Fraction Integer:=matrix[[1/(i+j+1) for j in 0..9] for i in 0..9]
```

$$\begin{bmatrix} 1 & \frac{1}{2} & \frac{1}{3} & \frac{1}{4} & \frac{1}{5} & \frac{1}{6} & \frac{1}{7} & \frac{1}{8} & \frac{1}{9} & \frac{1}{10} \\ \frac{1}{2} & \frac{1}{3} & \frac{1}{4} & \frac{1}{5} & \frac{1}{6} & \frac{1}{7} & \frac{1}{8} & \frac{1}{9} & \frac{1}{10} & \frac{1}{11} \\ \frac{1}{3} & \frac{1}{4} & \frac{1}{5} & \frac{1}{6} & \frac{1}{7} & \frac{1}{8} & \frac{1}{9} & \frac{1}{10} & \frac{1}{11} & \frac{1}{12} \\ \frac{1}{4} & \frac{1}{5} & \frac{1}{6} & \frac{1}{7} & \frac{1}{8} & \frac{1}{9} & \frac{1}{10} & \frac{1}{11} & \frac{1}{12} & \frac{1}{13} \\ \frac{1}{5} & \frac{1}{6} & \frac{1}{7} & \frac{1}{8} & \frac{1}{9} & \frac{1}{10} & \frac{1}{11} & \frac{1}{12} & \frac{1}{13} & \frac{1}{14} \\ \frac{1}{6} & \frac{1}{7} & \frac{1}{8} & \frac{1}{9} & \frac{1}{10} & \frac{1}{11} & \frac{1}{12} & \frac{1}{13} & \frac{1}{14} & \frac{1}{15} \\ \frac{1}{7} & \frac{1}{8} & \frac{1}{9} & \frac{1}{10} & \frac{1}{11} & \frac{1}{12} & \frac{1}{13} & \frac{1}{14} & \frac{1}{15} & \frac{1}{16} \\ \frac{1}{8} & \frac{1}{9} & \frac{1}{10} & \frac{1}{11} & \frac{1}{12} & \frac{1}{13} & \frac{1}{14} & \frac{1}{15} & \frac{1}{16} & \frac{1}{17} \\ \frac{1}{9} & \frac{1}{10} & \frac{1}{11} & \frac{1}{12} & \frac{1}{13} & \frac{1}{14} & \frac{1}{15} & \frac{1}{16} & \frac{1}{17} & \frac{1}{18} \\ \frac{1}{10} & \frac{1}{11} & \frac{1}{12} & \frac{1}{13} & \frac{1}{14} & \frac{1}{15} & \frac{1}{16} & \frac{1}{17} & \frac{1}{18} & \frac{1}{19} \end{bmatrix} \tag{1}$$

```
                                        Type: Matrix Fraction Integer
```

Figure 2.3: Hilbert Matrix.

lates SPAD code into COMMON LISP or into machine code. To be more precise, when a program is compilable, it is either compiled into COMMON LISP and then interpreted by the COMMON LISP interpreter, or it is further compiled from COMMON LISP to machine code. This depends on the overhead of further compilation during run-time. When AXIOM cannot completely determine the type of every object in each function, the program may still be executed in the interactive mode. In this mode, each statement is analyzed and executed as the control flow indicates.

The typical way of programming in AXIOM is as follows. The programmer creates an input file defining some functions for his or her application. Then, the programmer runs the file and tries the functions. Once everything works well, the programmer may want to add the functions to the local AXIOM library. To do so, the programmer needs to integrate his or her code in AXIOM type constructors and then invoke the compiler.

By definition, an AXIOM type constructor is a function that returns a type which can be either a *category*, a *domain*, or a *package*. Roughly speaking, a domain is a class of objects. For example, `Polynomial` domain denotes polynomials, `Matrix` domain denotes matrices. A category is a class of domains which have common

properties. For example, the AXIOM category `Ring` designates the class of all rings with units, any AXIOM domain that has this property belongs to the category `Ring`. The source code for the category `Ring` is shown below.

```
Ring(): Category == Join(Rng,Monoid,LeftModule(%)) with
      --operations
      characteristic: () -> NonNegativeInteger
        ++ characteristic() returns the characteristic of the ring
        ++ this is the smallest positive integer n such that
        ++ \spad{n*x=0} for all x in the ring, or zero if no such n
        ++ exists.
        -- We can not make this a constant, since some domains are
        -- mutable
      coerce: Integer -> %
        ++ coerce(i) converts the integer i to a member of
        ++ the given domain.
      unitsKnown
        ++ recip truly yields
        ++ reciprocal or "failed" if not a unit.
        ++ Note: \spad{recip(0) = "failed"}.
   add
      n:Integer
      coerce(n) == n * 1$%
```

From the above AXIOM source code we can observe another important concept: categories form hierarchies. We can see that `Ring` is extended from the categories `Rng`, `Monoid` and `LeftModule`. In addition, we can observe that `Ring` has

- 2 operations: `characteristic`, `coerce`,
- 1 attribute `unitsKnown`,
- and 1 default implementation for the operation `coerce: Integer -> %`.

The programmer can construct his (her) own categories by extending existing categories. This requires knowledge of the existing hierarchies. Figure 2.4 shows a fragment of the hierarchy of the AXIOM algebraic categories.

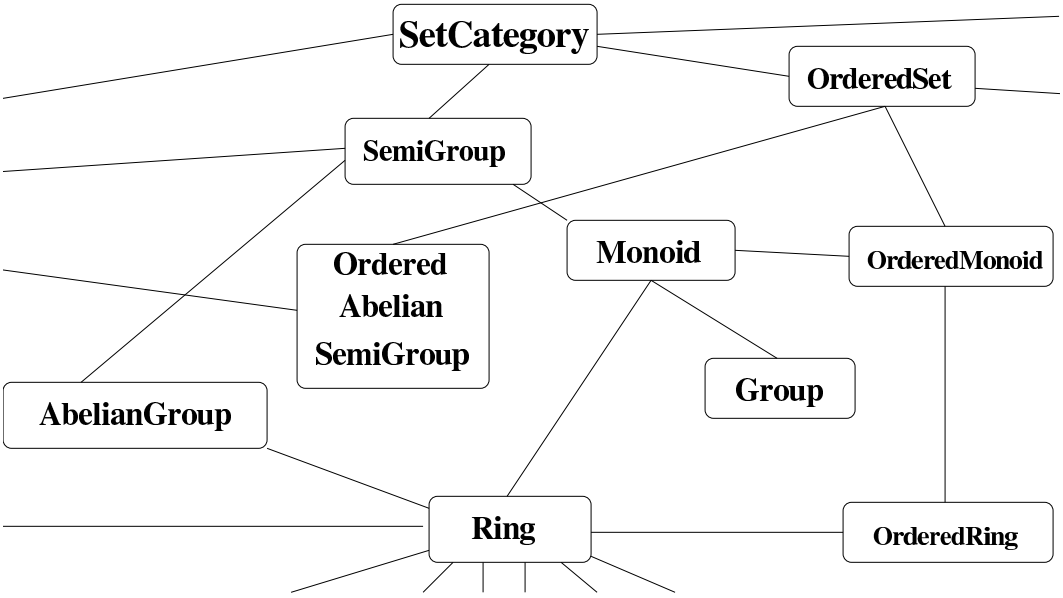

Figure 2.4: Algebraic category hierarchy in AXIOM (partial).

Next to the concept of category, domain is easier to understand. It actually corresponds to the notion of data type. When a domain is defined, it is asserted to belong to one or more categories and promises to implement the set of operations defined in these categories. After a newly defined domain is compiled, it becomes an AXIOM data type which can be used just like a system-provided data type. The programmer usually needs to design a lower level data structure to represent the objects of the domain. When a domain is instantiated, the AXIOM system will allocate memory for those data structures.

Packages are special cases of domains. Usually, a package is a place to hold a collection of polymorphic functions that work on many different domain types. In

other words, these polymorphic functions are generic functions that can accept any data type for one of its parameters. When a package is compiled, its functions are added to the local library. The user is responsible to load a compiled package into memory before using its functions.

2.3 The underlying LISP

As above mentioned, AXIOM has a very high-level programming language called SPAD. The AXIOM compiler translates SPAD code into COMMON LISP, then invokes the underlying LISP compiler to generate machine code. There are 3 distributions of COMMON LISP that have had AXIOM builds in the past. AXIOM was initially built on the CMUCL (Carnegie Mellon University COMMON LISP) distribution but was quickly moved to AKCL (Austin Kyoto COMMON LISP). Now GCL (GNU COMMON LISP) [1] is the underlying LISP of AXIOM. The design of GCL makes use of the native C compiler for compiling to native object code. This provides both good performance and portability [40].

2.4 The GMP library

GCL extensively makes use of functionality from the highly optimized GMP library (or GNU MP library). The GMP library is a portable library written in C for arbitrary precision arithmetic on integers, rational numbers, and floating-point numbers.

Many applications use just a few hundred bits of precision; but some applications may need thousands or even millions of bits. GMP is designed to give good performance for both, by choosing algorithms based on the sizes of the operands, and by carefully keeping the overhead at a minimum. The performance of GMP is achieved

- by using full machine words for every basic arithmetic type,
- by using fast algorithms,

- by including carefully optimized ASSEMBLY code,
- and by a general emphasis on speed (as opposed to simplicity or elegance).

2.5 Writing code at SPAD, LISP, C and ASSEMBLY levels

As above mentioned, the SPAD code is translated into LISP code by the SPAD compiler, then translated into C code by the GCL compiler. Finally, GCL makes use of a native C compiler, such as GCC, to generate machine code. Since those compilers can generate fairly efficient code, programmers can concentrate on their mathematical algorithms and write them in SPAD. However, to achieve higher performance, our implementation also involves lower level development, see Figure 2.5. Our reasons are as follows.

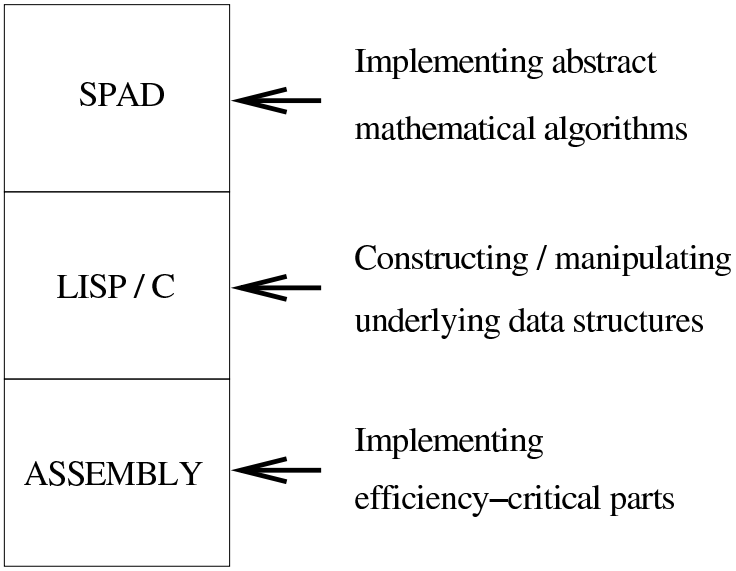

Figure 2.5: Language levels in AXIOM.

- We are curious about how exactly a compiler can optimize our code, and what it cannot do for us.

- Our work is largely motivated by modular methods. High performance for these methods relies on appropriately utilizing machine arithmetic as well as carefully constructing underlying data structures. This requires us to look into machine-level stuff, such as machine integer arithmetic, memory hierarchy, and processor architecture. At this level, C and ASSEMBLY code is preferred.

- We are also interested in parallel programming which is not available at SPAD level. Instead, it can be achieved in LISP and C. We will explain where we used parallelism programming in Chapter 5. Another reason for our LISP level implementation is that, we are willing to directly manipulate these LISP level data structures rather than those at the AXIOM level. This will avoid some potential overhead.

Certainly, we are not going to implement too many things at the intermediate or low level. In fact, AXIOM's great strength has been in research and development of mathematical algorithms. Thus, we have implemented all the high-level abstract and generic mathematical algorithms in AXIOM, more precisely, in the SPAD language.

Chapter 3

Adapted Data Structures I

In order to manipulate mathematical objects in a computer algebra system, we must represent them in memory by means of data structures. For example, in AXIOM, a square matrix M of order n with coefficients in some ring R is represented by a $n \times n$ two-dimensional array with entries in R. This representation is called *dense*, since all the coefficients of M are encoded. Other representations can be used for matrices, such as sparse representations, where only the non-zero coefficients are encoded.

In this chapter, we review sparse and dense representation for univariate and multivariate polynomials. Then, we describe their implementation in our AXIOM code combining the LISP and the SPAD levels. We also explain the C and ASSEMBLY level implementation for univariate polynomials over finite fields, as presented in Chapter 4. Finally, we report on experiments that illustrate the benefits of our multi-level software architecture.

3.1 Polynomial data representation

The source code of each AXIOM domain defines a special local variable, called `Rep`. Its value is the data structure used for representing the objects of this domain. The following code shows a fragment of the source code of the domain constructor `SparseUnivariatePolynomial`, abbreviated as `SUP`.

```
1 SparseUnivariatePolynomial(R: Ring): UnivariatePolynomialCategory(R)
2  with
3    outputForm: (%, OutputForm)-> OutputForm
4       ++ outputForm(p, var) converts the SparseUnivariatePolynomial
5       ++ p to an output form printed as a polynomial in the output
6       ++ form variable.
7    fmecg: (%, NonNegativeInteger, R, %) -> %
8       ++ fmecg(p1, e, r, p2) finds X: P1 - r*X**e*p2
9  == PolynomialRing(R, NonNegativeInteger)
10  add
11    -- representations
12    Term := Record(k: NonNegativeInteger, c:R)
13    Rep := List Term
```

For a given AXIOM ring `R`, this domain `SUP(R)` implements the ring `R[x]` of univariate polynomials with coefficients in `R`. The variable `x` is not specified since `R[x]` and `R[y]` are clearly isomorphic. The representation of these polynomials is *sparse*, which means that only non-zero terms are encoded. More precisely,

- each term is encoded as a record consisting of a non-negative integer, the *degree or the exponent of the term*, and an element of `R`, the *coefficient of the term*;
- a polynomial `p` of `SUP(R)` is represented by a list of terms such that the following conditions hold:
  - (*i*) none of these terms has a zero coefficient,
  - (*ii*) two different terms have different degrees,
  - (*iii*) the sum of these terms equals `p`.

It follows that the null polynomial of `R[x]` is encoded by the empty list.

In addition, the representation of the elements of `SUP(R)` is canonical, which means that there is a one-to-one map between the ring `R[x]` and the representation of the objects of `SUP(R)`. For a given polynomial `p` of `SUP(R)`, this *canonicity* is

achieved by sorting the list of its terms w.r.t their degrees. The `SUP` constructor chooses to sort terms by decreasing degrees. Hence, the leading coefficient and the degree of `p` are respectively the coefficient and the degree of the first term of `p`. As shown in the left frame of Figure 3.1, where `R` is for instance the ring $\mathbb{Z}$ of the integers, the univariate polynomial $7x^4 + 3$ has $((4\ \ 7)(0\ \ 3))$ as its sparse representation. Another possible representation for the univariate polynomials of `R[x]` is the *dense* one where a polynomial `p` is given by a record consisting of

- a non-negative integer `d`, which is the degree of `p`,
- an array `a` of size `d + 1`, called the *coefficient-array* of `p`, such that its i-th entry is the coefficient of x^i in `p`, for all $0 \leq i \leq$ `d`.

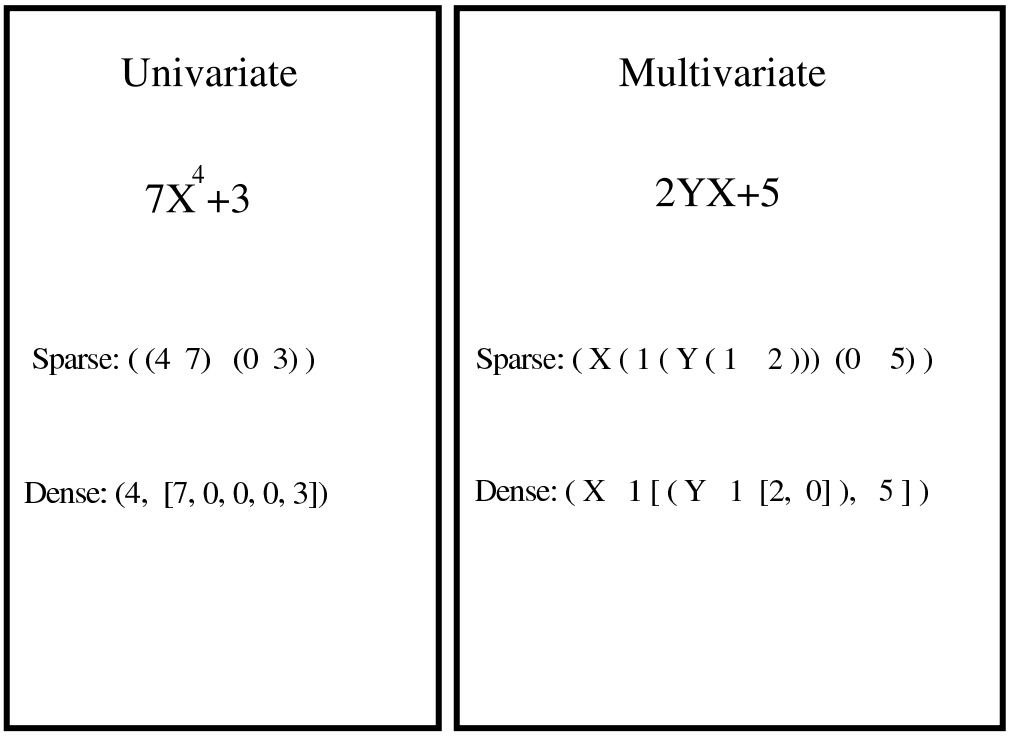

Figure 3.1: Sparse and dense polynomial representation.

In the left frame of Figure 3.1, we show the dense representation of $7x^4 + 3$. Observe that, for convenience, the entries of the coefficient-array for `p` are shown by

decreasing index from left to right. Other representations are used for univariate polynomials, such as the *expression tree* of the MAPLE computer algebra system. We refer to [18] and [41] for comprehensive discussions of polynomial representations and their implementations in a computer algebra system.

Both the sparse and the dense representations of univariate polynomials can be used recursively for implementing multivariate polynomials. Indeed, bivariate polynomials of `R[x,y]` can be viewed as univariate polynomials of `R[y][x]` or `R[x][y]`. In the right frame of Figure 3.1, we show the sparse and the dense representations of the bivariate polynomial $2\,x\,y + 5$ regarded as a polynomial of `R[y][x]`. Observe that the univariate representations require a slight modification in order to be used for representing multivariate polynomials: they need to keep track of the variable in the representation.

Each of these representations (sparse, dense, expression-tree) of univariate or multivariate polynomials has its strengths and drawbacks. Let us look at two extreme situations for illustrating this fact. Consider first a multivariate polynomial p_n with many variables $X_1, \ldots, X_n$, say $n = 100$, high degrees w.r.t. each of them, say $d = 100$ again, but few terms, say

$$p_n = \begin{cases} X_n^d\, p_{n-1} + 1 & \text{if} \quad n > 1 \\ X_1^d + 1 & \text{if} \quad n = 1 \end{cases} \tag{3.1}$$

The size of the dense representation of p is in the order of $O(n\,d)$ elements of the coefficient ring, whereas the size of the sparse representation is only in $O(d)$.

Consider now a univariate polynomial u of high degree, say $d = 1,000,000$, and with coefficients in $\mathbb{Z}/p\mathbb{Z}$ (The field of integers modulo the prime number p) for a small prime p which fits in a machine word. If there is a coefficient at each degree, the sparse representation of u would consist of a list of d records which may be scatteredly allocated in memory. Instead, the dense representation of p could be a block of d consecutive words in memory, allowing much faster access to a given coefficient.

In order to continue our comparison between the spare and the dense representations, consider the implementation of polynomial addition. Here again, we consider

two extreme situations. First, assume that f and g are two univariate polynomials such that g consists of a single term of degree greater than that of f. In the sparse case, one can implement $g + f$ just by creating a list with the term of g as "CAR" and the terms of f as "CDR", which amounts to $O(1)$ memory allocations. In the dense case, one needs to allocate a new array of size $\deg(g) + 1$.

Assume now that our polynomials f and g have large degrees and many non-zero coefficients. In the sparse case, the implementation of $f + g$ follows the scheme of a list merging algorithm. In addition, it requires many memory allocations of "small" pieces of memory (one per new created term). In the dense case, once the coefficient-array of the result has been allocated, the implementation of $f+g$ consists a straight forward loop to walk through the terms, something of the form:

```
for i in 0..n repeat
        r.i := f.i + g.i
```

(r is the sum of f and g) which may clearly run faster than the addition in the sparse case. Moreover, the dense representation makes the management of the memory more efficient.

Therefore, the choice between the sparse and the dense representations depends on the application. If the polynomials involved in some computation remain sparse, that is, if they have few non-zero terms compared to their degrees, then there is no point in using a dense representation. On the contrary, if the input polynomials are dense, or if the computations will "densify" the intermediate polynomials, it makes more sense to use the dense representation.

A typical algorithm which tends to densify the computations is the Euclidean Algorithm which is an algorithm for finding the greatest common divisor of two univariate polynomials over a field(or two integer numbers). Consider for instance the following input polynomials

$$r_0 = x^{200} + x^{10} + 1 \quad \text{and} \quad r_1 = x^{80} + x^8 + 1 \tag{3.2}$$

in the ring $\mathbb{Q}[x]$. The successive remainders of the Euclidean Algorithm applied to r_0 and r_1 are

- $r_2 = x^{56} + 2\ x^{48} + x^{40} + x^{10} + 1$
- $r_3 = 5\ x^{48} + 4\ x^{40} - x^{34} + 2\ x^{26} - x^{24} - 3\ x^{18} + 2\ x^{16} + 4\ x^{10} - 2\ x^8 + 5$
- $r_4 = \frac{1}{5}\ x^{42} + \frac{1}{25}\ x^{40} - \frac{4}{25}\ x^{34} + \frac{1}{5}\ x^{32} + \frac{3}{25}\ x^{26} - \frac{4}{25}\ x^{24} - \frac{2}{25}\ x^{18} - \frac{2}{25}\ x^{16} + \frac{1}{25}\ x^{10} - \frac{13}{25}\ x^8 - \frac{1}{5}$
- $r_5 = \frac{1001}{625}\ x^{40} - \frac{29}{25}\ x^{38} + \frac{29}{125}\ x^{36} - \frac{154}{625}\ x^{34} - \frac{74}{125}\ x^{32} + \frac{23}{25}\ x^{30} - \frac{23}{125}\ x^{28} + \frac{273}{625}\ x^{26} + \frac{121}{625}\ x^{24} + \frac{8}{25}\ x^{22} - \frac{8}{125}\ x^{20} - \frac{367}{625}\ x^{18} + \frac{123}{625}\ x^{16} + \frac{66}{25}\ x^{14} - \frac{66}{125}\ x^{12} + \frac{566}{625}\ x^{10} - \frac{263}{625}\ x^8 + x^6 - \frac{1}{5}\ x^4 + \frac{1}{25}\ x^2 + \frac{124}{125}$
- $r_6 = \frac{525625}{1002001}\ x^{38} + \frac{20000}{1002001}\ x^{36} - \frac{3750}{13013}\ x^{34} + \frac{768750}{1002001}\ x^{32} - \frac{416875}{1002001}\ x^{30} - \frac{166875}{1002001}\ x^{28} + \frac{2500}{11011}\ x^{26} - \frac{101250}{91091}\ x^{24} - \frac{145000}{1002001}\ x^{22} + \frac{404375}{1002001}\ x^{20} - \frac{184375}{1002001}\ x^{18} - \frac{2166250}{1002001}\ x^{16} - \frac{108750}{91091}\ x^{14} - \frac{23750}{91091}\ x^{12} - \frac{60000}{1002001}\ x^{10} - \frac{2987500}{1002001}\ x^8 - \frac{453125}{1002001}\ x^6 + \frac{90625}{1002001}\ x^4 - \frac{643750}{1002001}\ x^2 - \frac{1575625}{1002001}$
- $r_7 = \frac{511511}{707281}\ x^{36} - \frac{1439438}{707281}\ x^{34} + \frac{1088087}{707281}\ x^{32} + \frac{7007}{24389}\ x^{30} - \frac{558558}{707281}\ x^{28} + \frac{1924923}{707281}\ x^{26} - \frac{861861}{707281}\ x^{24} - \frac{19019}{24389}\ x^{22} + \frac{634634}{707281}\ x^{20} + \frac{2466464}{707281}\ x^{18} - \frac{525525}{707281}\ x^{16} + \frac{10010}{24389}\ x^{14} - \frac{420420}{707281}\ x^{12} + \frac{4358354}{707281}\ x^{10} - \frac{2640638}{707281}\ x^8 - \frac{5005}{24389}\ x^6 + \frac{30030}{24389}\ x^4 + \frac{1477476}{707281}\ x^2 - \frac{1178177}{707281}$
- $r_8 = \frac{1396901}{261121}\ x^{34} - \frac{1306073}{261121}\ x^{32} - \frac{31117}{37303}\ x^{30} - \frac{252300}{261121}\ x^{28} - \frac{2249675}{261121}\ x^{26} + \frac{140447}{37303}\ x^{24} + \frac{407044}{261121}\ x^{22} - \frac{1982237}{261121}\ x^{20} - \frac{487780}{37303}\ x^{18} - \frac{65598}{37303}\ x^{16} - \frac{802314}{261121}\ x^{14} - \frac{248936}{37303}\ x^{12} - \frac{102602}{5329}\ x^{10} + \frac{2434695}{261121}\ x^8 - \frac{458345}{261121}\ x^6 - \frac{1977191}{261121}\ x^4 - \frac{1869543}{261121}\ x^2 + \frac{932669}{261121}$
- $r_9 = \frac{1451751}{2758921}\ x^{32} + \frac{785918}{2758921}\ x^{30} + \frac{494137}{2758921}\ x^{28} + \frac{91469}{2758921}\ x^{26} - \frac{1803830}{2758921}\ x^{24} + \frac{222796}{250811}\ x^{22} + \frac{2809989}{2758921}\ x^{20} + \frac{1538110}{2758921}\ x^{18} - \frac{2954091}{2758921}\ x^{16} + \frac{2029692}{2758921}\ x^{14} + \frac{1194718}{2758921}\ x^{12} + \frac{40369}{2758921}\ x^{10} - \frac{4301598}{2758921}\ x^8 + \frac{1421091}{2758921}\ x^6 + \frac{1051638}{2758921}\ x^4 - \frac{811468}{2758921}\ x^2 - \frac{2893282}{2758921}$
- $r_{10} = \frac{2444992}{8071281}\ x^{30} + \frac{2089538}{8071281}\ x^{28} - \frac{2219096}{8071281}\ x^{26} - \frac{22750717}{8071281}\ x^{24} + \frac{6844981}{8071281}\ x^{22} + \frac{1019854}{2690427}\ x^{20} + \frac{9319871}{8071281}\ x^{18} - \frac{12728243}{2690427}\ x^{16} + \frac{597960}{896809}\ x^{14} - \frac{486673}{8071281}\ x^{12} - \frac{4801951}{8071281}\ x^{10} - \frac{9713528}{2690427}\ x^8 + \frac{352132}{896809}\ x^6 + \frac{191015}{896809}\ x^4 - \frac{1376969}{8071281}\ x^2 - \frac{18359033}{8071281}$
- $r_{11} = \frac{821049}{541696}\ x^{28} + \frac{1230153}{135424}\ x^{26} - \frac{7537173}{1083392}\ x^{24} + \frac{1423341}{1083392}\ x^{22} - \frac{804003}{541696}\ x^{20} + \frac{19361415}{1083392}\ x^{18} - \frac{9889521}{1083392}\ x^{16} + \frac{309669}{135424}\ x^{14} + \frac{2951799}{1083392}\ x^{12} + \frac{12275961}{1083392}\ x^{10} - \frac{1082421}{135424}\ x^8 + \frac{184665}{270848}\ x^6 + \frac{71025}{47104}\ x^4 + \frac{7338303}{1083392}\ x^2 - \frac{4707537}{1083392}$

- $r_{12} = \frac{2879600}{83521}\ x^{26} - \frac{2819432}{83521}\ x^{24} + \frac{687608}{83521}\ x^{22} - \frac{1300512}{83521}\ x^{20} + \frac{5881560}{83521}\ x^{18} - \frac{4015064}{83521}\ x^{16} + \frac{681168}{83521}\ x^{14} + \frac{130456}{83521}\ x^{12} + \frac{3484776}{83521}\ x^{10} - \frac{3296176}{83521}\ x^{8} + \frac{218224}{83521}\ x^{6} + \frac{112424}{83521}\ x^{4} + \frac{2110296}{83521}\ x^{2} - \frac{1857480}{83521}$

- $r_{13} = \frac{489294629}{244922500}\ x^{24} - \frac{84853001}{244922500}\ x^{22} + \frac{15566407}{122461250}\ x^{20} - \frac{51711059}{48984500}\ x^{18} + \frac{848011833}{244922500}\ x^{16} - \frac{11383999}{61230625}\ x^{14} + \frac{66509593}{244922500}\ x^{12} + \frac{44817253}{244922500}\ x^{10} + \frac{322278061}{122461250}\ x^{8} - \frac{7199857}{61230625}\ x^{6} - \frac{2507653}{244922500}\ x^{4} + \frac{1074213}{244922500}\ x^{2} + \frac{79874687}{48984500}$

- $r_{14} = \frac{101577545700}{2866455549721}\ x^{22} + \frac{367076384000}{2866455549721}\ x^{20} - \frac{333608859000}{2866455549721}\ x^{18} + \frac{272633704600}{2866455549721}\ x^{16} + \frac{73493401600}{2866455549721}\ x^{14} + \frac{181230098700}{2866455549721}\ x^{12} - \frac{95624598700}{2866455549721}\ x^{10} - \frac{70116632400}{2866455549721}\ x^{8} + \frac{95985644200}{2866455549721}\ x^{6} + \frac{93781623400}{2866455549721}\ x^{4} - \frac{233932944800}{2866455549721}\ x^{2} + \frac{36036190800}{2866455549721}$

- $r_{15} = \frac{179398565293636}{10531900693521}\ x^{20} - \frac{54943352709941}{3510633564507}\ x^{18} + \frac{117687363770051}{10531900693521}\ x^{16} + \frac{9087736866857}{10531900693521}\ x^{14} + \frac{27503113958149}{3510633564507}\ x^{12} - \frac{29314013696810}{10531900693521}\ x^{10} - \frac{7870248236452}{3510633564507}\ x^{8} + \frac{27346896912740}{10531900693521}\ x^{6} + \frac{61025918153225}{10531900693521}\ x^{4} - \frac{95570962105003}{10531900693521}\ x^{2} + \frac{2527399767739}{1170211188169}$

- $r_{16} = \frac{2517066778702173}{11227749627077776}\ x^{18} - \frac{3818112212630217}{11227749627077776}\ x^{16} + \frac{381679883443605}{11227749627077776}\ x^{14} - \frac{1538892758851425}{11227749627077776}\ x^{12} - \frac{388249516683645}{5613874813538888}\ x^{10} - \frac{691018345032231}{2806937406769444}\ x^{8} - \frac{60451575413454}{701734351692361}\ x^{6} - \frac{963937271272515}{11227749627077776}\ x^{4} - \frac{170313672155631}{11227749627077776}\ x^{2} - \frac{2469351761856789}{11227749627077776}$

- $r_{17} = \frac{849182055193466800}{601565221018811449}\ x^{16} + \frac{343702214828429548}{601565221018811449}\ x^{14} + \frac{682224379091263268}{601565221018811449}\ x^{12} + \frac{673295990111439292}{601565221018811449}\ x^{10} + \frac{547091169006122260}{601565221018811449}\ x^{8} + \frac{460333296636481632}{601565221018811449}\ x^{6} + \frac{383125215937085648}{601565221018811449}\ x^{4} + \frac{294034834904863988}{601565221018811449}\ x^{2} + \frac{429248529314696936}{601565221018811449}$

- $r_{18} = \frac{8093646079436159529}{64225707449292490000}\ x^{14} + \frac{8963514770014081139}{64225707449292490000}\ x^{12} + \frac{36664367930801793141}{64225707449292490000}\ x^{10} - \frac{5166279318582404449}{12845141489858498000}\ x^{8} + \frac{1655921845497232367}{8028213431161561250}\ x^{6} + \frac{553008257063777544}{4014106715580780625}\ x^{4} + \frac{5923554723478841499}{64225707449292490000}\ x^{2} - \frac{311063035823698661}{32112853724646245000}$

- $r_{19} = -\frac{3210613002134277836 00}{108894438325788541209}\ x^{12} + \frac{260178685392579511400}{36298146108596180403}\ x^{10} - \frac{50329696078066373200}{15556348332255505887}\ x^{8} + \frac{6523576485821115 4700}{108894438325788541209}\ x^{6} + \frac{17696479989822163900}{36298146108596180403}\ x^{4} + \frac{34027149119020814900}{36298146108596180403}\ x^{2} + \frac{49162422013935572500}{108894438325788541209}$

- $r_{20} = \frac{48291151005537817316 31}{401242596572878762276}\ x^{10} - \frac{27570782657675374791 45}{401242596572878762276}\ x^{8} + \frac{20231586819690001624 89}{80248519314575752455 2}\ x^{6} + \frac{16019984539675401147 03}{80248519314575752455 2}\ x^{4} + \frac{16130722224868083211 89}{80248519314575752455 2}\ x^{2} + \frac{37313898450474624738 9}{80248519314575752455 2}$

- $r_{21} = -\frac{37302462364041459452987}{21415558969712598439392 9} x^8 + \frac{29418692208521123843 53}{14277039313141732292928 6} x^6 - \frac{10204521350404552579523}{42831117939425196878785 8} x^4 - \frac{19661683201604262591215}{42831117939425196878785 8} x^2 - \frac{34803628802314892079373}{42831117939425196878785 8}$
- $r_{22} = \frac{10625161139814314894463 99}{55486579353465726911265604} x^6 - \frac{19844309975589758372477 07}{55486579353465726911265604} x^4 - \frac{99988178500516085183449 35}{55486579353465726911265604} x^2 + \frac{13859935515210895384665945}{55486579353465726911265604}$
- $r_{23} = \frac{67553029541578289076276832}{52715901278450400631509 69} x^4 + \frac{64692822808128420125817 80}{17571967092816800210503 23} x^2 - \frac{11783640725899377196972 8352}{52715901278450400631509 69}$
- $r_{24} = -\frac{14489328729366219403529 3991}{20560880907683469568857664} x^2 + \frac{23864545606303620039559371}{25701101134604336961072 08}$
- $r_{25} = \frac{14749541926460186164326 40872}{39824918466007220836990 67449}$
- $r_{26} = 0$

The same observation is true for the "variants" of the Euclidean Algorithm such as the Extended Euclidean Algorithm or the Subresultant-GCD Algorithm [17, 18].

Another major observation on the above example is the swell of the size the intermediate expressions. This problem is usually overcome by the so-called *modular methods*. The standard modular approach for the above GCD computation would be to

- compute the gcd g_i of the input polynomials r_0 and r_1 modulo several primes p_i, for $i = 1, 2, 3, \ldots$,
- keeping only those g_i with minimum degree,
- recombining them by the *Chinese Remainder Algorithm* into a polynomial h,
- until $h = 1$ or until h divides both the input polynomials r_0 and r_1,
- then return h.

See [17, 18] for a precise algorithm.

The "densification" will happen also for the Euclidean Algorithm applied to $\mathbb{Z}/p\mathbb{Z}[x]$ for a prime integer p. With the previous input polynomials, consider $p = 9001$. Then we have

- $r_2 = x^{56} + 2x^{48} + x^{40} + x^{10} + 1$

- $r_3 = 5x^{48} + 4x^{40} + 9000x^{34} + 2x^{26} + 9000x^{24} + 8998x^{18} + 2x^{16} + 4x^{10} + 8999x^8 + 5$
- $r_4 = 7201x^{42} + 8641x^{40} + 1440x^{34} + 7201x^{32} + 7921x^{26} + 1440x^{24} + 720x^{18} + 720x^{16} + 8641x^{10} + 4680x^8 + 1800$
- $r_5 = 7188x^{40} + 1439x^{38} + 6913x^{36} + 5818x^{34} + 5328x^{32} + 721x^{30} + 1656x^{28} + 6870x^{26} + 1858x^{24} + 6121x^{22} + 576x^{20} + 7085x^{18} + 29x^{16} + 3243x^{14} + 4752x^{12} + 4451x^{10} + 1987x^8 + x^6 + 1800x^4 + 8641x^2 + 73$
- $r_6 = 3530x^{38} + 2810x^{36} + 7811x^{34} + 5623x^{32} + 8374x^{30} + 7495x^{28} + 7211x^{26} + 4411x^{24} + 2130x^{22} + 1121x^{20} + 2786x^{18} + 2801x^{16} + 4071x^{14} + 4924x^{12} + 571x^{10} + 1053x^8 + 4406x^6 + 919x^4 + 7439x^2 + 1962$
- $r_7 = 4399x^{36} + 6574x^{34} + 5271x^{32} + 6433x^{30} + 5906x^{28} + 1036x^{26} + 2822x^{24} + 541x^{22} + 2710x^{20} + 1730x^{18} + 6331x^{16} + 189x^{14} + 6865x^{12} + 2341x^{10} + 1757x^8 + 4406x^6 + 567x^4 + 6735x^2 + 5844$
- $r_8 = 606x^{34} + 2338x^{32} + 274x^{30} + 7824x^{28} + 6757x^{26} + 1196x^{24} + 2859x^{22} + 2424x^{20} + 8674x^{18} + 8119x^{16} + 3998x^{14} + 2192x^{12} + 729x^{10} + 1907x^8 + 8813x^6 + 6588x^4 + 5410x^2 + 4561$
- $r_9 = 6115x^{32} + 8161x^{30} + 5652x^{28} + 2489x^{26} + 2759x^{24} + 61x^{22} + 6618x^{20} + 5297x^{18} + 2736x^{16} + 4795x^{14} + 2691x^{12} + 5423x^{10} + 2877x^8 + 4275x^6 + 105x^4 + 2307x^2 + 3385$
- $r_{10} = 8204x^{30} + 1190x^{28} + 2142x^{26} + 6505x^{24} + 979x^{22} + 8554x^{20} + 2875x^{18} + 6473x^{16} + 7687x^{14} + 1669x^{12} + 6023x^{10} + 6515x^8 + 6427x^6 + 6331x^4 + 7487x^2 + 8033$
- $r_{11} = 2411x^{28} + 6881x^{26} + 3058x^{24} + 2666x^{22} + 2342x^{20} + 8759x^{18} + 8138x^{16} + 6347x^{14} + 1235x^{12} + 536x^{10} + 7873x^8 + 6971x^6 + 3473x^4 + 3720x^2 + 1513$
- $r_{12} = 1261x^{26} + 6474x^{24} + 1965x^{22} + 7050x^{20} + 3746x^{18} + 6858x^{16} + 4743x^{14} + 7247x^{12} + 8181x^{10} + 3045x^8 + 5648x^6 + 2424x^4 + 3162x^2 + 5309$
- $r_{13} = 5906x^{24} + 347x^{22} + 8788x^{20} + 3709x^{18} + 6619x^{16} + 7975x^{14} + 7395x^{12} + 6255x^{10} + 8367x^8 + 4149x^6 + 3955x^4 + 3618x^2 + 7512$

- $r_{14} = 4750x^{22} + 5939x^{20} + 2877x^{18} + 5760x^{16} + 8653x^{14} + 8488x^{12} + 4333x^{10} + 3981x^8 + 6301x^6 + 5558x^4 + 809x^2 + 4201$
- $r_{15} = 4336x^{20} + 4x^{18} + 1182x^{16} + 1663x^{14} + 624x^{12} + 5944x^{10} + 305x^8 + 3078x^6 + 8685x^4 + 1334x^2 + 2936$
- $r_{16} = 6051x^{18} + 551x^{16} + 415x^{14} + 1415x^{12} + 4790x^{10} + 8468x^8 + 2415x^6 + 7982x^4 + 6827x^2 + 4702$
- $r_{17} = 6024x^{16} + 5394x^{14} + 4088x^{12} + 1448x^{10} + 6181x^8 + 4176x^6 + 8815x^4 + 5325x^2 + 5532$
- $r_{18} = 7303x^{14} + 2528x^{12} + 2944x^{10} + 4521x^8 + 6050x^6 + 3804x^4 + 1252x^2 + 5950$
- $r_{19} = 5410x^{12} + 1514x^{10} + 3601x^8 + 420x^6 + 93x^4 + 6855x^2 + 7966$
- $r_{20} = 7529x^{10} + 7116x^8 + 1531x^6 + 7869x^4 + 924x^2 + 3800$
- $r_{21} = 8684x^8 + 3438x^6 + 3046x^4 + 2263x^2 + 5606$
- $r_{22} = 3412x^6 + 6417x^4 + 5021x^2 + 5406$
- $r_{23} = 518x^4 + 7371x^2 + 3257$
- $r_{24} = 860x^2 + 7762$
- $r_{25} = 2682$
- $r_{26} = 0$

Therefore, one should use a dense representation for implementing the Euclidean Algorithm. Moreover, in the case of $\mathbb{Z}/p\mathbb{Z}[x]$, the coefficients have a bounded size. For instance, if p fits in a machine word, like above, one can implement the coefficient-array of every polynomial $f \in \mathbb{Z}/p\mathbb{Z}[x]$ of degree d using an array of $d+1$ machine words. This gives an additional reason for using the dense representation in this case.

3.2 Dense recursive multivariate polynomials at the SPAD level

When we started our work, the AXIOM libraries (written in SPAD) provided only sparse (univariate and multivariate) polynomials. In our experiments, we used the AXIOM domain constructors

- `SparseUnivariatePolynomial`, presented above,
- `SparseMultivariatePolynomial`, abbreviated as `SMP`, which provides sparse multivariate polynomials viewed as univariate polynomials, implemented by means of `SUP`.

AXIOM provides other sparse multivariate polynomials. For instance, the domain constructor `DistributedMultivariatePolynomial` which implements multivariate polynomials as lists of terms, where each term consist of a coefficient and a multivariate monomial. See [22] for more details.

We have implemented a domain constructor, called `DenseUnivariatePolynomial`, abbreviated as `DUP`, for univariate polynomials in dense representation over an arbitrary AXIOM ring. Based on the `DUP`, we have implemented a constructor, called `DenseRecursiveMultivariatePolynomial`, abbreviated as `DRMP`, for multivariate polynomials in dense representation over an arbitrary AXIOM ring.

For a given AXIOM ring `R`, the domains `SUP(R)` and `DUP(R)` provide exactly the same operations, that is they have the same user interface, which is defined by the category `UnivariatePolynomialCategory(R)`. But, of course, the implementation of the operations of `SUP(R)` and `DUP(R)` is quite different, as we saw in the case of the addition. In `SUP(R)`, we are mainly dealing with the data structure "Linked List". On the other hand, in `DUP(R)` we are primarily working with the data structure "Array". Similarly, the domains `SMP(R)` and `DRMP(R)` both satisfy `PolynomialCategory(R)`. See [22] for more details.

Another major concern with our implementation of the constructors `DUP` and `DMPR` is to minimize memory consumption. For instance, destructive operations,

that is, operations which recycle the memory allocated to some of their arguments, are used intensively. One typical example for this is polynomial division. Consider two univariate polynomials $f = \Sigma_0^n\, a_i x^i$ and $g = \Sigma_0^m\, b_i x^i$ of positive degrees and such that the leading coefficient b_m of g is invertible. The quotient q and the remainder r in the division of f by g can be computed by the following standard algorithm

1 $n < m \Rightarrow$ **return** $(0, f)$
2 $r := f$
3 **for** $i = n - m, n - m - 1, \ldots, 0$ **repeat**
4 **if** $\deg r = m + i$ **then**
5 $q_i :=$ leadingCoefficient$(r)\,/\,b_m$
6 $r := r - q_i x^i g$
7 **else** $q_i := 0$
8 $q := \Sigma_0^{n-m}\, q_i x^i$
9 **return** (q, r)

Since the degree of the polynomial r decreases during the execution of the algorithm one can avoid memory allocation in Step (6) and recycle the space allocated to r. Of course, if we do not want to destroy f we must replace Step (2) by $r :=$ **copy** f.

This technique of *memory recycling* is straightforward to implement in the case of the polynomial division. Consider now the Karatsuba algorithm for polynomial multiplication [24]. This is a divide-and-conquer algorithm that we recall below for the case $\deg(f) = \deg(g) = n$. (One can easily reduce the general case to this case by adding a "zero leading coefficient" to the polynomial of smallest degree.) It performs $O(n^{\log_2(3)})$ operations of the coefficient ring, which is better than the classical quadratic multiplication algorithm.

1 **if** $n = 1$ **then return** $a_0 b_0$
2 $p := \lceil n/2 \rceil$
3 $(f_1, f_0) :=$ **divide**(f, x^p)
4 $(g_1, g_0) :=$ **divide**(g, x^p)
5 $low := f_0\, g_0$

6 $high := f_1\, g_1$

7 $middle := (f_1 - f_0)(g_1 - g_0)$

8 $middle := middle - high - low$

9 **return** $high\, x^{2p} + middle\, x^p + low$

Observe that

- Steps (3) and (4) do not need a polynomial division; for each of the polynomials f and g, they separate the terms of degree less than p from the others, such that we have $f = f_1 x^p + f_0$ and $g = g_1 x^p + g_0$ with $\deg(f_0) < p$ and $\deg(g_0) < p$.
- Steps (5), (6) and (7) are three recursive calls.

A direct implementation of this algorithm may lead to many memory allocations, for creating the polynomials f_1, f_0, g_1, g_0. In our implementation, we do not create these polynomials explicitly. Instead, we use "markers" (or pointers) in the data representation of the input polynomials f and g. Note that we do not destroy any data, we just avoid unnecessary data duplication.

3.3 Dense recursive multivariate polynomials at the LISP level

The domain constructors `SMP` and `DMP` allow the user to construct multivariate polynomials over any AXIOM ring. For this reason, we say that their SPAD code is generic.

AXIOM supports *conditional implementation* in the following sense: if the coefficient ring has some properties then a specialized algorithm can be called to speed up a given operation. For instance, if the coefficient ring is $\mathbb{Z}/p\mathbb{Z}$, for a prime p, one can use formulas such *Little Fermat Theorem* to speed up exponentiation.

Ideally, one would like to use also *conditional data representations*. For instance, if the coefficient ring is $\mathbb{Z}/p\mathbb{Z}$, for a **small** prime p, one would like to use arrays of machine words for encoding univariate polynomials over $\mathbb{Z}/p\mathbb{Z}$, as discussed earlier.

But the code would become quite complicated, (then harder to optimize from the compilation point of view) since many tests would be needed for selecting the appropriate representation. Moreover, the special case of the coefficient ring $R = \mathbb{Z}/p\mathbb{Z}$ is so important (for modular methods) that it deserves an independent treatment.

For these reasons, we have realized a special implementation of dense multivariate polynomials over $\mathbb{Z}/p\mathbb{Z}$, for a prime p (large or small). These polynomials are implemented at the LISP level which offers us more flexibility (less type checking, better support from the machine arithmetic) than at the SPAD level, which is a strongly-typed language.

Since we restrict ourselves to the coefficient ring $R = \mathbb{Z}/p\mathbb{Z}$, we can think of improving the data representation presented earlier. Indeed, Let `R` be an AXIOM ring. Recall that a polynomial f of `SMP(R)` and `DMPR(R)` is viewed as a univariate polynomial constructed by `SUP` and `DUP` respectively. More precisely, a coefficient of f is either another polynomial or an element of `R`. At the SPAD level, this disjunction is implemented by a union type. In order to reduce the number of indirections, we follow the *vector-based approach* proposed by Richard J. Fateman [14] where a polynomial is either a number or a vector, as Figure 3.2 shows.

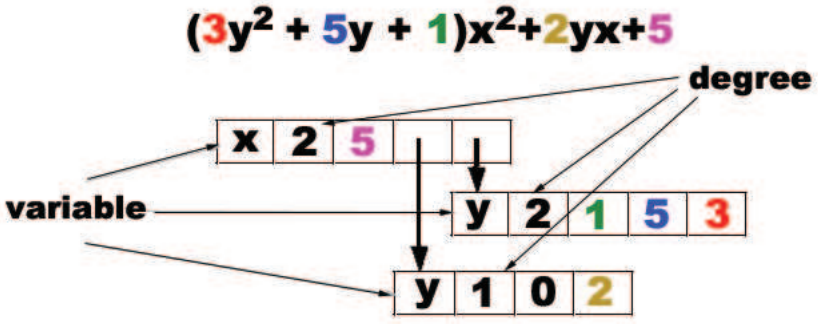

Figure 3.2: Vector-based multivariate polynomial recursive representation.

In this representation, each constant polynomial is represented by a number and each non-constant polynomial is represented by a vector. The first slot of a vector keeps the information of current main variable of this polynomial, and the second slot keeps the information of the degree of current polynomial respect to

main variable. In addition:

- If a coefficient is a polynomial, then the corresponding slot keeps a pointer pointing to that polynomial or, say, another vector.
- If a coefficient is a number, then the corresponding slot keeps the pointer to this number.

We implemented this data representation at LISP level, since we can directly use so-called predicate functions to judge the type of objects. This technique avoids union types and, thus, makes storage more compact. If we want to implement this data representation at SPAD level, we have to wrap up those predicate functions in LISP and make them available for SPAD. This will introduce extra overhead. Moreover, if p fits in a machine word, we will tell the compiler to use machine integer arithmetic. This technique yields significant speed up, as reported in the next section.

3.4 Experimentation

In order to compare our different polynomial constructors, we choose the algorithm of van Hoeij and Monagan [20]. We recall its specifications. Let $\mathbb{K} = \mathbb{Q}(a_1, a_2, \ldots, a_e)$ be an algebraic number field over the field $\mathbb{Q}$ of the rational numbers. An example of such field is

$$\mathbb{Q}(\sqrt{2}, \sqrt{3}) = \{a + b\sqrt{2} + c\sqrt{3} + d\sqrt{2}\sqrt{3} \mid a, b, c \in \mathbb{Q}\}. \tag{3.3}$$

Let $f_1, f_2 \in \mathbb{K}[y]$ be univariate polynomials over $\mathbb{K}$. The algorithm of van Hoeij and Monagan computes the $\gcd(f_1, f_2)$ by using a modular method instead of the Euclidean Algorithm.

Figure 3.3 gives the principle of the algorithm of van Hoeij and Monagan, which is very similar to the standard modular approach given above for polynomial GCDs in $\mathbb{Q}[y]$. Figure 3.4 shows the complete flowchart of the algorithm of van Hoeij and Monagan.

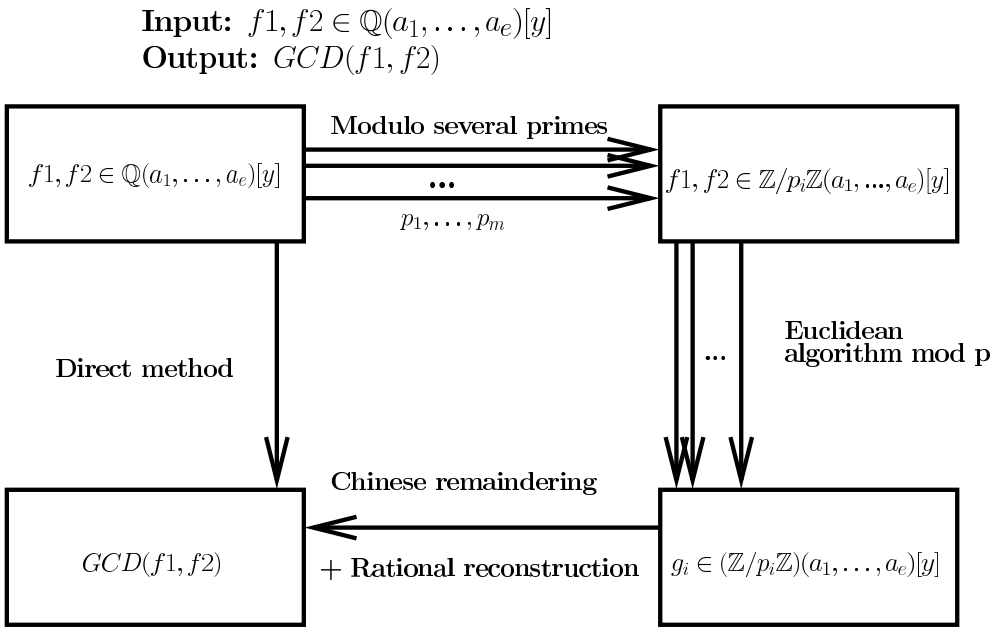

Figure 3.3: Principle of the algorithm of van Hoeij and Monagan.

The algebraic expressions in $\mathbb{K}$ are encoded with multivariate polynomials. Therefore, the algorithm involves two polynomial types:

- a multivariate polynomial type for the coefficients of f_1 and f_2, which are in $\mathbb{K}$,
- a univariate polynomial type for f_1 and f_2 themselves.

As shown on Figure 3.5 p. 35 and Figure 3.6 p. 36, we have used the following combinations:

$\mathbb{Q}(a_1, a_2, \ldots, a_e)$	+	$\mathbb{K}[y]$
SMP in SPAD	+	SUP in SPAD
DMPR in SPAD	+	DUP in SPAD
Dense in LISP	+	SUP in SPAD
Dense in LISP	+	DUP in SPAD

Observe that

- the first two combinations, that is SMP + SUP and DMPR + DUP involve only SPAD code,

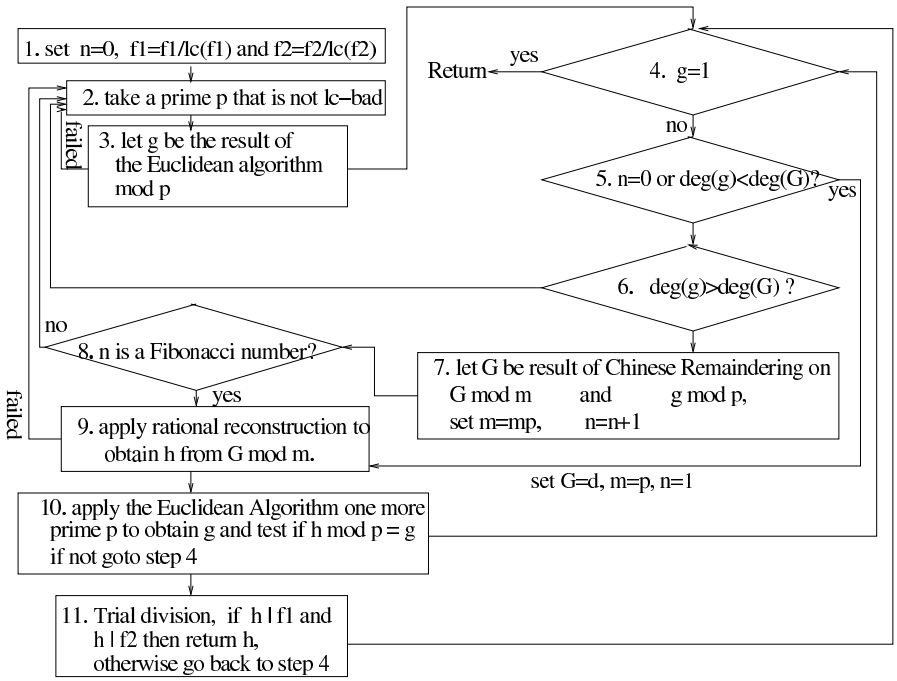

Figure 3.4: The flowchart of van Hoeij and Monagan's algorithm.

- the other two use our dense multivariate polynomials implemented at the LISP level and univariate polynomials at the SPAD level.

At this point, we would like to stress the following facts

- the algorithms for addition, multiplication, division of our `DMPR` and our LISP-level dense polynomials are identical,
- none of the above polynomial types uses fast arithmetic (that is algorithms based on the FFT).

Remember also that

- the SPAD constructors `SMP`, `DMPR`, `SUP`, and `DUP` are generic constructors, i.e. they work over any AXIOM ring,
- however, our dense multivariate polynomials implemented at the LISP level only work over a prime field, that is a field of the form $\mathbb{Z}/p\mathbb{Z}$ where p is a prime number.

Therefore, we are comparing here is the performance of

- sparse representation versus dense representation when computations densify,
- specialized code at the LISP level versus generic code at the SPAD level.

The benchmarks reported on Figure 3.5 and Figure 3.6 show that

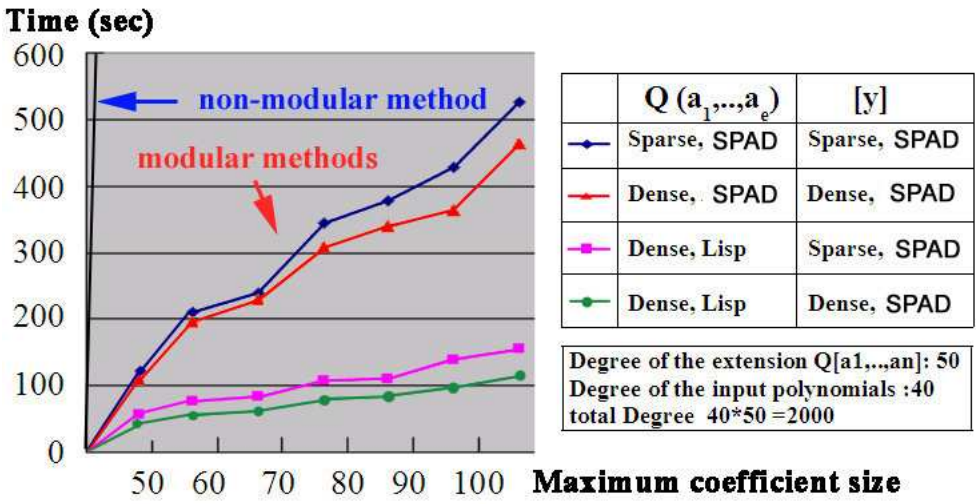

Figure 3.5: Comparison for a fixed total degree and increasing input coefficient size.

- dense representation in $\mathbb{K}[y]$ gives a slight improvement w.r.t. sparse representation,
- specialized dense multivariate polynomials lead to a dramatic improvement w.r.t. generic multivariate polynomials.

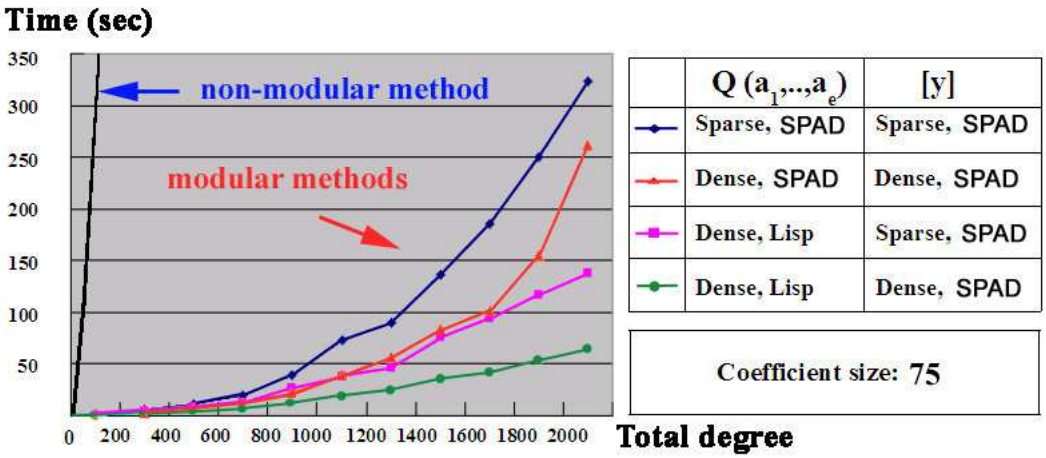

Figure 3.6: Comparison for a fixed input coefficient size and increasing total degree.

Chapter 4

Adapted Data Structures II

4.1 Modular methods and fast arithmetic

Symbolic computations manipulate numbers by using their mathematical definitions rather than using floating point approximations. Consequently, their results are exact, complete and can be made canonical. However, they can be huge! Moreover, intermediate expressions may be much bigger than the input and output.

Modular methods help keep the swell of the intermediate expressions under control. In broad words, a modular method for computing some quantity q in a polynomial ring $R[X_1, \ldots, X_n]$ is an indirect way which

- performs the computations in $R[X_1, \ldots, X_n]/\mathcal{I}_1$, ..., $R[X_1, \ldots, X_n]/\mathcal{I}_s$, for some ideals $\mathcal{I}_1, \ldots, \mathcal{I}_s$ in R, obtaining quantities, $q_1, \ldots, q_s$,
- and constructs q from $q_1, \ldots, q_s$.

For a general discussion on modular methods, please refer to [17].

In this thesis, we restrict ourselves to modular methods where R is the ring $\mathbb{Z}$ of the integer numbers and the ideals $\mathcal{I}_1, \ldots, \mathcal{I}_s$ are generated by prime numbers $p_1, \ldots, p_s$.

The simplest and most famous example of modular computation is that of the determinant. Consider a square matrix A of order n, with integer entries. Let $B > 0$ be an upper bound for the absolute values of the entries of A and let C be an upper

bound for the absolute value of its determinant d. For instance, we can use the well-known Hadamard bound:

$$C = n^{n/2} B^n. \tag{4.1}$$

Consider prime numbers $2 \leq p_1 < \cdots < p_s$ such that their product m is larger than $2C$. Let A_i be the image of A modulo p_i and let d_i be the determinant of A_i, for all $i = 1, \ldots, s$. It is easy to show that d_i is also the image of d modulo p_i, for all $i = 1, \ldots, s$. Let d' be the image of d modulo m obtained by the *Chinese Remaindering Algorithm* from $d_1, \ldots, d_s$, such that $-\frac{m-1}{2} \leq d' \leq \frac{m-1}{2}$ holds. Then, it is easy to show that $d' = d$ holds. This approach has the following advantages:

- the running time complexity is better than with a direct exact method, say Gaussian elimination,
- the size of the intermediate computations is kept under control, since most numbers are in one of the fields $\mathbb{Z}/p_1\mathbb{Z}, \ldots, \mathbb{Z}/p_s\mathbb{Z}$,
- if the prime numbers $p_1, \ldots, p_s$ are small, we can use machine arithmetic.

See [17, 18] for the details on the complexity analysis of this approach.

On top of the Chinese Remaindering Algorithm, there is also another popular way of reconstructing the desired result, namely Hensel lifting, which uses only one prime number p_1. Figure 4.1 shows the scheme of a modular method using only one prime. Again, we refer to [17, 18] for more details.

Quite often, modular methods lead to perform computations in a polynomial ring of the form $\mathbb{Z}/p\mathbb{Z}[X_1, \ldots, X_n]$, where p is a prime number. This has an additional advantage: for "good primes" p, one can use fast arithmetic, that is algorithms for multiplication, division, polynomial GCD that are nearly optimal. These algorithms

- are generally based on the Fast Fourier Transform (FFT, for short),
- and generally reduce multivariate computations to univariate ones.

See chapter 6 and 7 for examples of fast algorithms over prime fields. In addition, see [17] for an extensive discussion on fast arithmetic.

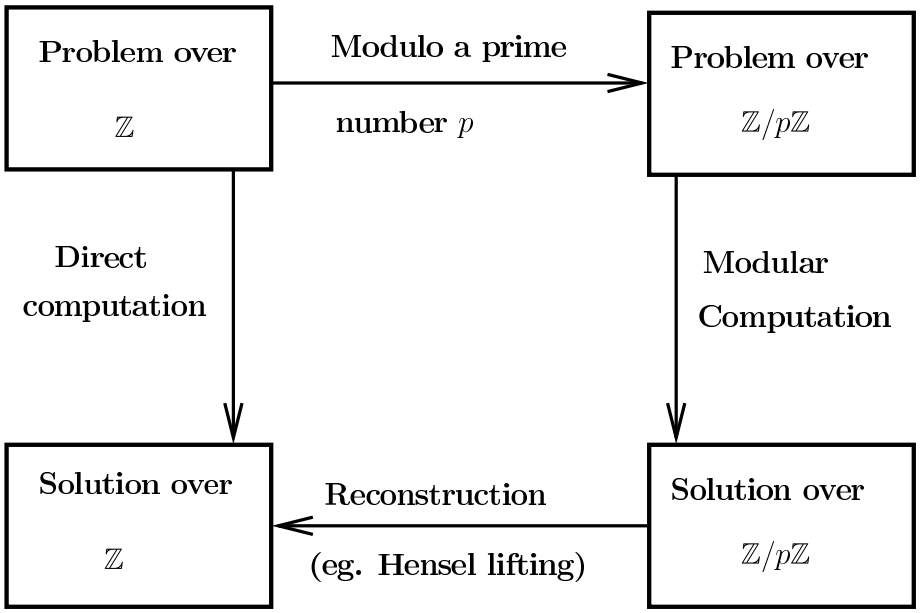

Figure 4.1: Sketch of a modular method using only one prime.

Therefore, it is a main concern to realize a highly efficient implementation of the univariate polynomial rings $\mathbb{Z}/p\mathbb{Z}[X]$. This is the topic of this chapter. Two cases need to be discussed:

- p is a small prime, that is, p fits in a machine word,
- p is a large prime.

The first case can obviously be optimized. But the second case is important for situations where small primes cannot be used.

4.2 Univariate polynomials over $\mathbb{Z}/p\mathbb{Z}$ as machine integer arrays

The dense polynomial representation is suitable for the class of algorithms in which polynomials are generally densified such as the Euclidean Algorithm. This is also true for our modular methods, which usually make use of variants of the Euclidean Algorithm. As discussed in the previous chapter, a dense univariate polynomial is

represented by an array where each entry keeps a coefficient. If coefficients belong to a prime field $\mathbb{Z}/p\mathbb{Z}$, then they are encoded by an integer in the range $0 \cdots p-1$. If the prime number p is small, then every coefficient fits in a machine word. This property allows us to use C-like arrays to encode the polynomials of $\mathbb{Z}/p\mathbb{Z}[x]$.

As discussed in Chapter 2, the SPAD language is a LISP-based language. So, a SPAD array type is actually a LISP array type. The underlying GCL provides us with a special array type called `fixnum-array`. Following code shows the definition of `fixnum-array` in GCL.

```
1   struct fixarray{
2                   FIRSTWORD;
3           object  fixa_displaced; /* displaced */
4           short   fixa_rank;      /* array rank */
5           short   fixa_elttype;   /* element type */
6           fixnum  * fixa_self;    /* pointer to the array */
7           short   fixa_adjustable /* adjustable flag */
8           short   fixa_offset;    /* not used */
9           int     fixa_dim;       /* dimension */
10          int     * fixa_dims;    /* table of dimensions */
11  };
```

The field `fixa_self` is a pointer. The user can allocate a block of memory with consecutive entries of `fixnum`, then let this pointer point to this block of memory. Using `fixnum-array` for representing polynomials of $\mathbb{Z}/p\mathbb{Z}[x]$ has three benefits.

Fast memory allocation. Like other GCL data types, `fixnum-array` uses the GCL allocator system to allocate memory. Our experimentation results show that GCL's allocator is faster than `malloc`, the default allocater in C. So, we usually let GCL create a new `fixnum-array` instead of invoking `malloc` in C.

Mimicking big integers. Both Euclidean domains $\mathbb{Z}$ and $\mathbb{K}[x]$, where $\mathbb{K}$ is a field, share many properties. The implementation of $\mathbb{Z}$ and that of dense univariate

polynomials over a field (in particular over a prime field) are also similar (see Chapter 2 in [17]). Indeed, both can be encoded with radix representations. Hence, adapting a code for big integers in order to obtain a code for univariate polynomials over a prime field is a natural idea. Observe also that Kronecker's substitution can be used in order to perform univariate polynomial multiplication via big integer multiplication.

GC friendly. GCL's garbage collector is a tracing GC which traces reachable objects by pointers. Because a `fixnum-array` has no pointers within its data fields, it can be recycled easily.

Figure 4.2 gives an example of a coefficient array implemented with `fixnum-array`, in the small prime case. The big prime case deserves its own section, which follows.

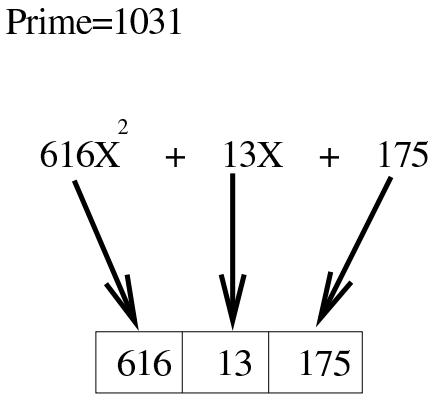

Figure 4.2: Encoding a univariate polynomial over $\mathbb{Z}/m\mathbb{Z}$ in fixnum-array, small prime case.

4.3 Big prime case

Most CAS provide multi-precision number arithmetic, also called big integer arithmetic. Such numbers can be arbitrarily big as long as there is sufficient memory to encode them.

Let f and g be two univariate polynomials in $\mathbb{Z}[x]$ such that every coefficient of f, g, $f+g$, fg lies in the range $-m \cdots m$ where numbers can be encoded by pairwise different machine words. Then:

- each of f and g can be encoded by an array of machine words, and
- each of $f + g$ and fg can be computed exactly (that is without losing any precision) by means of machine arithmetic.

Clearly, this will be faster than using big integer arithmetic for the coefficients of f, g, $f + g$, fg. Ideally, we would like to turn all polynomial computations into univariate polynomial computations over a small prime field in order to minimize the use of big integer arithmetic. Unfortunately, this is impossible. Indeed, some modular methods, for instance for solving systems of non-linear multivariate equations, require the use of big primes, since small primes would not be able to preserve enough information from the input system and, thus, would make reconstruction unsuccessful [10].

Currently, the open source AXIOM, and more precisely, the underlying GCL employs the GNU Multi-Precision library (GMP) [16] for its big integer arithmetic. So, every AXIOM integer is a GMP integer. (Former versions of AXIOM, in particular the ones commercialized by NAG, did not have these properties.)

As mentioned above, for the modular methods considered in this thesis, all integer coefficients of a polynomial are bounded by a prime number p. Based on this observation, we still use `fixnum-array` as the data representation for dense univariate polynomials with coefficients in $\mathbb{Z}/p\mathbb{Z}$ for a big prime p. To do this, we just use $\lfloor \log_M(p) \rfloor + 1$ machine words per coefficient, where M is the largest positive integer which can be encoded in a machine word. Figure 4.3 shows a polynomial over such a prime field, encoded with a `fixnum-array`. Besides faster allocation and easier garbage collection, can this data representation help us write more efficient code? The answer is yes. Using this data representation, we can accomplish tasks, such as univariate polynomial addition, in ASSEMBLY code in a very efficient manner. In fact, we just pass the pointers of our coefficient array to our ASSEMBLY functions

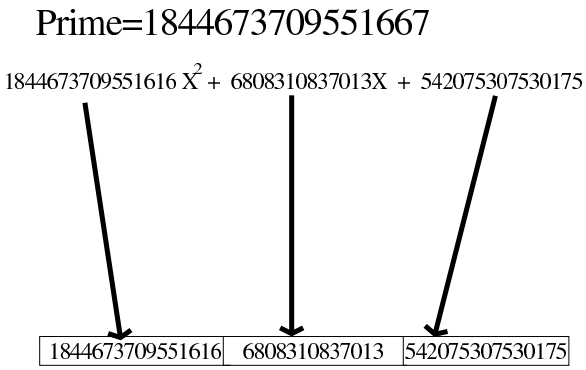

Figure 4.3: Encoding a univariate polynomial over $\mathbb{Z}/p\mathbb{Z}$ with a `fixnum-array`, in the big prime case.

and return the result to AXIOM (as shown in the Figure 4.4).

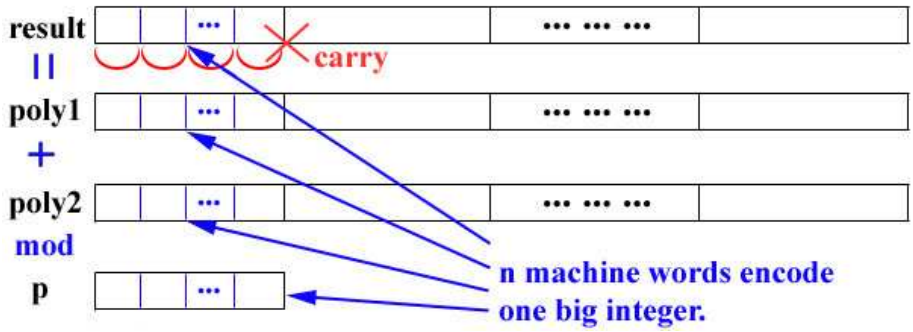

Figure 4.4: Polynomial addition over $\mathbb{Z}/p\mathbb{Z}$.

Figure 4.5 and Figure 4.6 show benchmarks between

- The `SUP` constructor (from the SPAD level, described in Chapter 3) instantiated over a big prime field
- `UMA`, our dense univariate polynomials written in LISP, C and ASSEMBLY code.

In the case of univariate polynomials over $\mathbb{Z}/p\mathbb{Z}$, for a big prime p, Figure 4.5 and Figure 4.6 clearly illustrate the benefits of our implementation combining LISP, C and ASSEMBLY code.

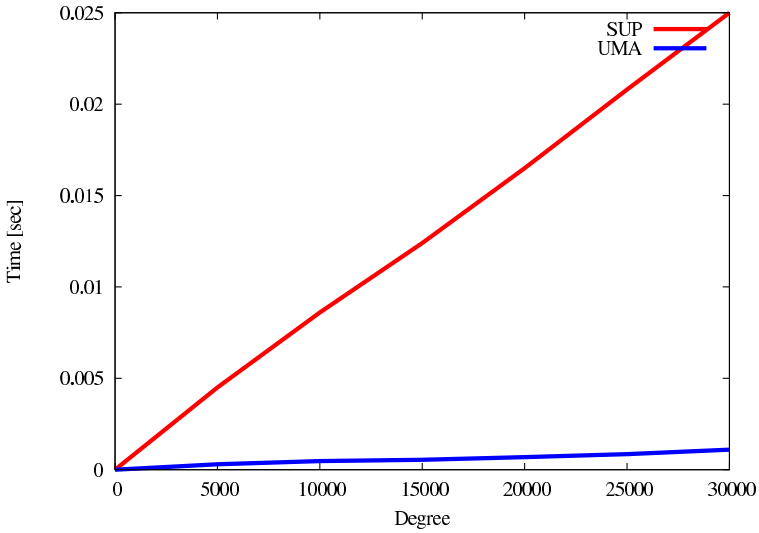

Figure 4.5: Polynomial addition over $\mathbb{Z}/p\mathbb{Z}$ in the big prime case.

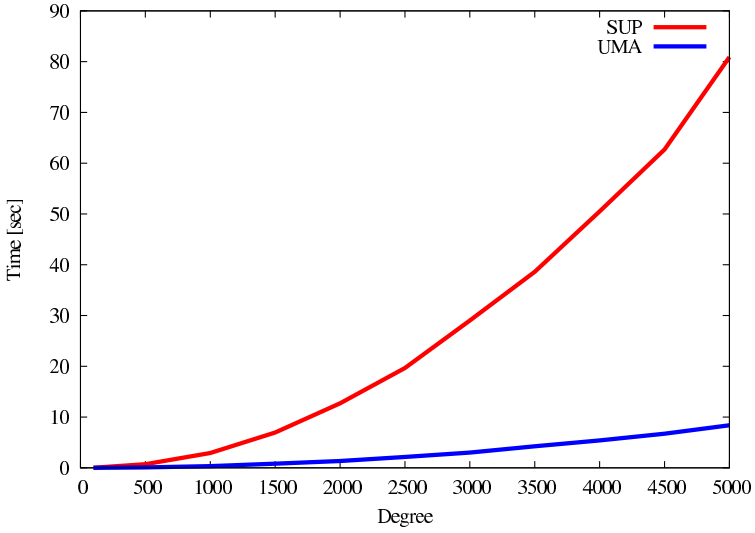

Figure 4.6: Polynomial multiplication over $\mathbb{Z}/p\mathbb{Z}$ in the big prime case.

Chapter 5

Highly optimized low level routines

5.1 Introduction

As previously discussed, selecting adapted data structures and fast algorithms is one of the dominant factors for high performance of exact polynomial computations. In this chapter, we would like to focus on another factor which is less obvious but indispensable: understanding the efficiency of compiled code.

Traditionally, the main responsibility of a compiler is to translate programs from source language to native machine language. This process is called code generation. Nowadays, a compiler's development work is more focused on code optimization. In other words, the compiler tries to remove the inefficiency introduced by source programs and optimize the performance of the compiled code (primarily speeding up the code). It's undeniable that many modern compilers can generate highly efficient code even if a source program is not written in a performance-oriented way. However, this not saying we can ignore the basic knowledge of compilation, if we are pursuing the goal of high performance. The reasons are as follows.

First, compilers cannot understand the program's behavior as clearly as programmers do. They usually optimize the code in a very precautious way so as not to change any correct code behavior. A highly complicated program will constrain

the compiler to apply some more aggressive optimization techniques. Therefore, it is the programmer's responsibility to write neat code which is easier for a compiler to understand and optimize. For our code, we check the compiled code to understand which kind of optimization techniques the compiler already applied and which additional improvement we can implement. Then, we tune our source code for the compiler to better optimize it, and modify the compiled code if necessary.

Second, some compilers do not always use all the latest features of a computer architecture. One possible reason is that some new features do not have a wide applicability (they are only attractive to some special usage). Another reason is that the compiler has not been updated to use those new features, yet. Let us quote Randal E. Bryant and David O'Hallaron " a number of formats and instructions have been added to IA32 for manipulating vectors of small integers and floating point numbers... Unfortunately, current versions of GCC will not generate any code that uses these new features" [3].

Therefore, if programmers want to enhance the performance further, they need to consider "machine dependent" implementations besides the generic versions. This requires to study modern computer architecture. In this case, programmers can directly make use of any feature in a hardware system, even if the compiler has no knowledge about it.

During our development, we first try to write "compiler friendly code", then implement the optimizations that the compiler cannot do for us. To make sure that our efforts lead indeed to performance improvement rather than degradation, we have benchmarked any modified version with the pervious one. These comparisons may also inspire new ideas for later development work.

In this chapter we will focus on discussing our C and ASSEMBLY level implementations which support our AXIOM level implementation. At the C level, we care about how to write compiler friendly code. At the ASSEMBLY level, we have implemented architecture specific routines which the GCC compiler cannot generate. In the following sections, we will give specific examples to illustrate the ideas mentioned in this introduction.

5.2 Single precision integer division by floating-point arithmetic

Our fast polynomial arithmetic is mainly based on modular methods to reduce the size of problems. Modular methods use modular arithmetic or, say remainder arithmetic, at each step. Therefore, the most frequently used operation in our implementation is integer division.

Unfortunately, integer division by its mathematical characteristics is a very expensive operation. Although hardware designers already put tons of efforts to refine division algorithms and organize the division hardware such as divisor registers, quotient registers and remainder registers in the most efficient way, division is still much slower than multiplication, addition and subtraction. For example, one single precision integer division on a Pentium machine is several times slower than multiplication and tens of times slower than subtraction and addition [19]. So, improving the performance of integer division is one of the key issues in our implementation. Following [34], for the single precision integer division case, we replace the division by a combination of a few cheaper operations such as multiplication, addition, and subtraction.

For instance, a computation such as $(a + b) \mod p$ where a and b belongs to $\mathbb{Z}/\mathrm{p}\mathbb{Z}$ is easy to implement without integer division. Indeed, let us assume that the elements of $\mathbb{Z}/\mathrm{p}\mathbb{Z}$ are encoded by the integers in the interval $[0, p-1]$. (This is the case in our implementation.) So, for $0 \leq a, b < p$, if the sum $a + b$ exceeds $p - 1$ we subtract p from $a + b$.

Let us give another example which appears in the FFT of a univariate, see Chapter 6. For $0 \leq a, b, c < p$, assume that $a + bc \mod p$ is known and that we are looking for $a - bc \mod p$. The relation

$$a - bc \equiv 2a - (a + bc) \mod p$$

allows us to compute $a - bc \mod p$ via addition, subtraction and sign checking, avoiding again an integer division.

We also have saved some computational time by doing the division after several steps instead of at each step. For example, when we compute $(a + bc) \mod p$, we first compute $a + bc$ then we reduce the result modulo p. Our assembly code for this operation shows as follows:

```
1                   mull   %ebx
2                   addl   %edi, %eax
3                   jnc    L(nocarry)
4                   addl   $1, %edx
5       L(nocarry):
6                   divl   %ecx
```

In this code a, b, c are all machine integers. So, we know that by adding a to the product of bc, no carry is needed on the high word of this product.

There are also some special cases that allow us to do modular reductions without using division. For example, let $p > 2$ and let e be such that $2^e + 1 > p$ holds and p divides $2^e + 1$. Then, we have

$$2^e + 1 \equiv p - 1 \mod p.$$

Another frequent special case is when the divisor d (in an Euclidean division of some integer n by d) is a power of 2. Then, we can compute the remainder by only using shift and add/sub operations [37].

We implement the single precision modular reduction by means of floating point arithmetic, based on the following formula

$$a \equiv a - \lfloor a\ 1/p \rfloor\ p$$

The rational number $1/p$ is approximated by a floating point number (of type `double float` in C). Next, following the above formula, we compute two products and one subtraction instead of one division. Note that the final result needs to be Adjusted by adding p or subtracting p. This idea is already implemented in C++ by the NTL library.

With modular methods, a prime number p can be used through the whole algorithm or at least used repeatedly. Hence, we can assume that the inverse of p has been pre-computed.

In order to pursue even higher performance, we have also implemented this idea in ASSEMBLY language for the Pentium IA-32 with SSE2 support. SSE2 is an instruction set designed by Intel. Instructions in this set are all in the format of Single Instruction Multiple Data (SIMD) and their operands are eight 128-bit XMM registers. The set of instructions supports 64-bit double-precision floating point arithmetic (including some 64, 32, 16 and 8-bit integer operations). In fact, SSE2 is an extension of SSE (Streaming SIMD Extensions). The instruction set of SSE only supports 32-bit single-precision floating point operations which uses the same set of registers as SSE2. Before we talk about SSE/SSE2 further, let's see an example how GCC compiles C code that contains floating point arithmetic.

Here is a very simple C program. It computes the sum of two "double float" numbers:

```
1   main(){
2     double a=1.0, b=2.0, c;
3     c=a+b;}
```

By using the command " cc -S float.c ", we can check the Assembly code generated by GCC for "float.c".

```
1  .file "float.c"
2  .def ___main; .scl 2; .type 32; .endef
3  .section .rdata,"dr"
4  .align 8
5  LC1:
6  .long 0
7  .long 1073741824
8  .text
9  .globl _main
```

```
10  .def _main; .scl 2; .type 32; .endef
11   _main:
12  pushl %ebp
13  movl %esp, %ebp
14  subl $40, %esp
15  andl $-16, %esp
16  movl $0, %eax
17  addl $15, %eax
18  addl $15, %eax
19  shrl $4, %eax
20  sall $4, %eax
21  movl %eax, -28(%ebp)
22  movl -28(%ebp), %eax
23  call __alloca
24  call ___main
25  fld1
26  fstpl -8(%ebp)
27  fldl LC1
28  fstpl -16(%ebp)
29  fldl -8(%ebp)
30  faddl -16(%ebp)
31  fstpl -24(%ebp)
32  leave
33  ret
```

In the above ASSEMBLY code, we can observe that the generated code uses the traditional FPU instruction set, even on our machine, which has the most advanced SSE/SSE2 technology. Actually, neither the current version of Microsoft Visual C/C++ nor the GCC compiler makes use of SSE/SSE2 at all.

As we mentioned already, the SSE/SSE2 instruction sets use XMM registers. These registers are of 128 bits. Each register can pack 2 double precision floating

point numbers or 4 single precision floating point/integer numbers. This mechanism allows the SSE/SSE2 instructions to compute with multiple data packed in one register in parallel. This is the reason why Intel call these instructions SIMD. Consequently, if programmers know how to use these instructions, faster code may be achieved. This is what we have been doing in our implementation. Following code computes $(a * b) \mod p$ with SSE2 instructions.

```
1     movl PARAM_RPTR,%edx
2     movl PARAM_WD1,%eax
3     movl PARAM_WPD1,%ecx
4     movq (%edx),%mm0
5     movups (%eax),%xmm1
6     cvtpi2pd %mm0, %xmm0
7
8     movups (%ecx),%xmm2
9     movl PARAM_PD,%eax
10    mulpd %xmm0,%xmm1
11    mulpd %xmm0,%xmm2
12    movups (%eax),%xmm0
13    cvttpd2pi %xmm2, %mm2
14    cvtpi2pd %mm2, %xmm2
15
16    mulpd %xmm2,%xmm0
17    subpd %xmm0,%xmm1
18    cvttpd2pi %xmm1, %mm1
19    movq %mm1, (%edx)
20
21    emms
22    ret
```

In this example, we also used another set of registers MMX (MM0..MM7), so

called MultiMedia eXtensions. MMX instructions support 64-bit integer operations. One thing worthy to mention here is that each of the eight MMX 64-bit registers is physically equivalent to the low-order 64-bit of each of the FPU's registers. So it is not a good idea to mix FPU and MMX instructions in the same computation sequence.

Actually, in GMP, there is a set of library functions that heavily use MMX instructions to do big integer arithmetic, when they are available. We have also used the same strategy for our big prime arithmetic. We will discuss this in detail in Chapter 6.

Figure 5.1 illustrates the performance difference between

- the generic ASSEMBLY code using integer arithmetic and,
- the SSE2 ASSEMBLY code using floating point arithmetic.

The benchmark data of Figure 5.1 are obtained with our implementation of FFT-based univariate polynomial multiplication over $\mathbb{Z}/p\mathbb{Z}$.

This benchmark shows that our SSE2-based implementation is significantly faster than our generic ASSEMBLY version.

5.3 Reducing the overhead of loops and function calls

Many algorithms operating on dense polynomials have an iterative structure. One major overhead in implementing such algorithms is loop indexing and loop condition testing. We can reduce this overhead by replicating the loop body. This is a well known technique, called loop unrolling.

Optimizing compilers usually put a lot of their effort on loops and provide support for loop unrolling. For example, when we set up the flag "-funroll-loops" on the command line of GCC, it will perform loop unrolling automatically.

However, one thing is subtle: how many iterations a compiler will unroll for a loop? There is a tradeoff. On one hand, unrolled loops require less loop indexing

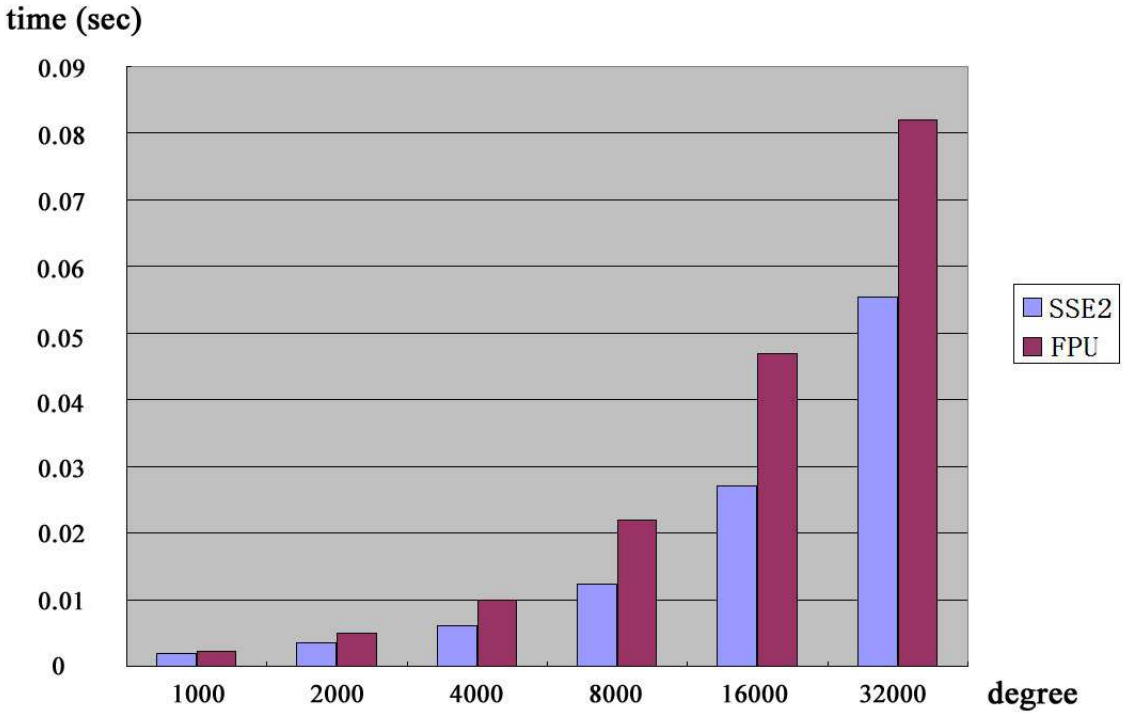

Figure 5.1: Generic assembly vs. SSE2 version assembly of FFT-based univariate polynomial multiplication over $\mathbb{Z}/p\mathbb{Z}$.

and less loop condition testing. On the other hand, unrolled loops suffer from code size growth. This will aggravate the burden of instruction caching. If the loop body consists of branching statement, increased number of branches in each iteration will have a negative impact on branch prediction.

Therefore, compilers usually do static or even run-time analysis to decide how much to unroll a loop. However, the analysis may not be precise when loops become complex and nested. Moreover, compilers are very precautious when removing small loops by unrolling them, since this may change the original program's data dependency. Moreover, optimizing compilers usually do not check if there is a possibility to combine unrolled statements together for better performance.

Based on these observations, in our implementation, we have unrolled some loop structures by hand, and recombined the related statements into small ASSEMBLY functions. This allows us to keep some values in registers or evict those unwanted

ones. The following code is a fragment of our implementation of the FFT-based univariate polynomial multiplication, detailed in Chapter 6.

```
#include "fftdfttab_4.h"
typedef void (* F) (long int *, long int,  long int,
                        long int *, long int, int);
typedef void (* G) (long int *, long int *,
                        long int *, long int, int);
inline void
fftdftTAB_4( long int * a, long int * b, long int * w,
                        long int p, F f, G g1, G g2 )
{
        long int w0=1, w4=w[4];
        long int * w8=w+8;
        f(a, w0, w4, a+2, p, 8);
        g2(a+4, w8, a+8, p, 4);
        g2(a+12, w8, a+16, p, 4);
        g1(a+8, w8, a+16, p, 8);
        f(b, w0, w4, b+2, p, 8);
        g2(b+4, w8, b+8, p, 4);
        g2(b+12, w8, b+16, p, 4);
        g1(b+8, w8, b+16, p, 8);
        return;
}
```

This function is dedicated to compute the case where $n = 4$ in the FFT algorithm. The functions `f()`, `g1()`, `g2()` are small ASSEMBLY functions which recombine related statements for higher efficiency. We also developed similar functions from the case $n = 5$ to the case $n = 8$. However, when n is bigger than 6, these specialized straight-line functions are less efficient than the version which uses nested loops. There are two main reasons. First, loop unrolling augments the size of binaries which

increases the cost of their management in the memory. Secondly, more instructions need to be cached, which increases memory traffic and degrades performances.

Figure 5.2 shows that for small degrees, the inlined version may gain about 10% running time. This is in fact a significant improvement. Indeed, our experiments show that 50% is already spent in performing the integer divisions.

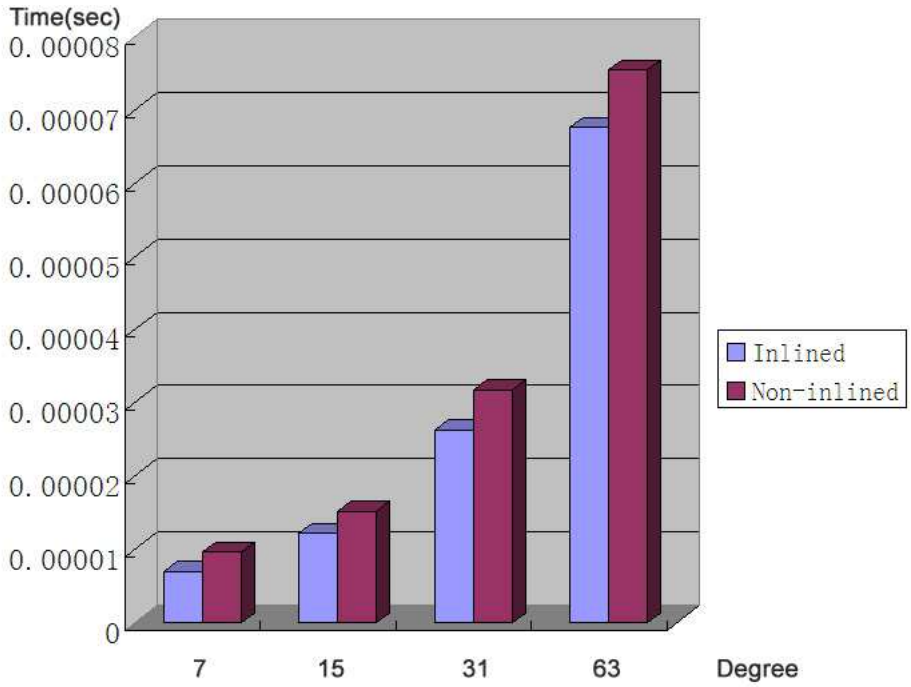

Figure 5.2: Inlined vs. non-inlined version of FFT-based univariate polynomial multiplication over $\mathbb{Z}/p\mathbb{Z}$.

Another frequently used technique in our implementation is function inlining. For instance, in our C code, by declaring `inline` a function `f`, we tell GCC to replace any call to `f` by the function body of `f`. This replacement happens during compilation, if we set up an appropriate optimization level. Obviously, function inlining will reduce the overhead caused by a function call, such as stack allocation and restoring the environment after this function call.

However, this is not saying that we can do function inlining as much as we want. Again, we need to consider the aspect of instruction cache. Too much function inlining may result in poor instruction cache behavior. To find the cutoff, we need

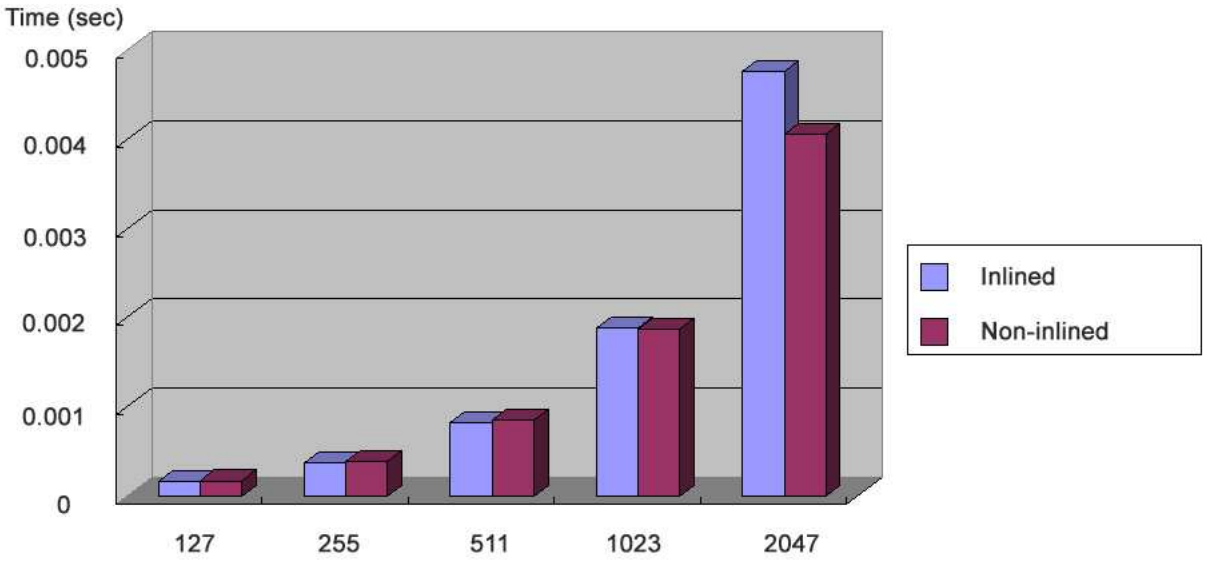

Figure 5.3: Inlined vs. non-inlined version of FFT-based univariate polynomial multiplication over $\mathbb{Z}/p\mathbb{Z}$.

to try a small test data set first and benchmark the impact after applying this to the whole application.

For both loop unrolling and function inlining, we should also consider the size of binary code. Larger binary code usually triggers more paging, and the data replacement between memory and secondary storage is extremely expensive. The Figure 5.3 shows that after some certain point, the inlined code becomes slower than the non-inlined code. Therefore, we only unroll small loops and inline small functions.

5.4 Memory traffic

One bottleneck of modern computer architectures is the mismatched speed between slower memory accessing and faster CPU processing. The usual case is when the CPU is idle for a while, waiting for data from memory. If we keep the data available as soon as the CPU needs it, we can speed up our applications significantly. This is especially true for our polynomial computations which access memory intensively.

We need to tune our programs for less memory traffic and better CPU usage. This requires us to understand the memory hierarchy.

In a modern computer architecture, CPU registers stay at the top level of the memory hierarchy, the cache system is at the middle level, main memory is at the low level and secondary storage such as hard disk is at the bottom level. The higher level is tens of times faster than the lower level.

According to the characteristics of this hierarchy, we use the following strategies to improve the performance of our implementation.

First, since we only have a very limited number of registers to use (for IA-32, we only have 4 general data registers, 5 general addressing registers, 8 floating point registers), we need to carefully design the algorithm and data structure for better utilizing the fastest and precious registers. We also need to avoid unnecessary exchange of data between registers and cache or main memory. One reason is that the exchanging operation itself is expensive. Another fact is that the absence of required data from registers will stall CPU pipeline.

Although optimizing compilers devote special efforts to make good use of the target machine's register set, this effort can be constrained by numerous factors, such as:

1. difficulty to estimate the execution frequencies of all parts of the program,
2. difficulty to allocate or evict ambiguous values,
3. difficulty to take advantage of some new hardware features on specific platforms.

Therefore, a high performance-oriented application requires programmers to better exploit the power of registers on a target machine. In fact, we have spent a great effort in this direction in our implementation.

First, we directly program the efficiency-critical parts in ASSEMBLY language in order to explicitly manipulate data in registers. For example, we write the classical univariate polynomial multiplication algorithm in both C and ASSEMBLY language,

where the polynomials are over $\mathbb{Z}/p\mathbb{Z}$ for a prime number p. The ASSEMBLY version is faster than the C version since we explicitly put the value of index variables in registers instead of a memory location. Although, the optimizing C compilers claim to do that for us, our benchmark results show that our ASSEMBLY implementation is faster than the C compiler compiled code. This avoids extra memory accessing, and saves many CPU cycles at each iteration. In C we can declare a variable `v` to be of "register" type in order to ask the compiler to put `v` into a register. However, defining such a register variable does not garantee that the register is reserved for this variable; the register remains available for other uses in places where flow control determines the value of `v` is not live. According to our benchmark result, our explicit register allocation method is always faster than the C compiler's optimization.

Besides generic assembly version of univariate polynomial addition, we also implemented the MMX-based version, where polynomials are over $\mathbb{Z}/p\mathbb{Z}$. In the latter version, we regard the polynomial coefficient array as a big integer, and use MMX registers to perform integer arithmetic. This not only reduces the number of machine operations, but it also reduces the frequency of memory accessing, since we have more registers to manipulate data.

We should be fully aware of one fundamental property of memory accessing —locality of reference. In fact, this property consists of two aspects: temporal locality and spatial locality. The temporal locality refers to the phenomenon that once one memory location is accessed, it is very likely to be accessed again in a very short time. Spatial locality refers to the phenomenon that once a memory location is accessed, its nearby locations are very likely to be accessed soon.

According to these observations, hardware designers introduced one type of small and fast memory called cache. Cache memory is located between CPU registers and main memory. The most recently used data and instructions are stored in cache, since they are likely to be reused in the near future. Nearby instructions and data are also constantly pre-fetched into cache for faster accessing when the CPU needs them. Because cache has very limited space, the least recently used data or instructions will be evicted into memory when new stuff comes in. After understanding this

hardware level design concept, we have tried to write cache-friendly code to speed up the whole applications.

As discussed in Chapter 3, our univariate polynomials over finite fields are encoded as machine integer arrays. For this kind of data structure, we can expect to write cache friendly code. One reason is that the coefficients of a polynomial are stored in a machine integer array and, thus, consecutively located in memory.

When we visit the terms of univariate polynomials by increasing (or decreasing) degree, we may hope to produce good code for spatial locality. Similarly, when we conduct several operations on a coefficient, we may hope to produce good code for temporal locality. However, this may not be sufficient as pointed by [23]. This is an area that we aim to further study, especially for the algorithms discussed in Chapters 6 and 7.

Usually, caching is processed by hardware. However, for some new hardware technology, such as Intel SSE/SSE2, several new instructions provide programmers with some level of control on cache. For example, if we know that some data or instructions will no longer be used,

- we can force them back to main memory without passing through the cache or,
- we can bring some data or instructions directly into registers without the risk of polluting the cache.

We experimented these new instructions in our implementation. Unfortunately, we didn't observe obvious performance improvement. We hope to find a more effective way of using those new features in later implementation work.

Finally, we should also consider the paging mechanism in the virtual memory system. For example, if a page fault occurs in a low-memory situation, the virtual memory system may need to reclaim some of the application's memory pages to make room for another process. Moreover, if these reclaimed pages have been modified, the system must first write the modifications to disk. As we know, writing data back to disk will consume a significant amount of time compared to memory access-

ing. Therefore, as programmer we should try to reduce the overhead introduced by paging. For instance, some data structures may be more suitable for implementing large polynomials when we consider the overhead of paging.

5.5 Parallelism

Parallelism is a fundamental technique used to achieve high speed on a modern computer architecture. There are two levels of parallelism: macroscopic and microscopic.

The examples of microscopic parallelism are within processors. As we know, a register is implemented by a separate digital circuit. Parallel hardware is used to move data between general-purpose registers and the ALU. Memory system also transfers data word by word instead of bit by bit. A typical modern computer has a bus that is either thirty-two or sixty-four bits wide, which means that thirty-two or sixty-four bits of data can be transferred across the bus in parallel.

Macroscopic parallelism refers to the use of parallelism across multiple large-scale components. For instance, dual processors or multiple processors can independently work in one computer. When more than one physical processor is available, computations can be conducted in parallel in each processor.

It is the compiler or programmer responsibility to discover the parallelizable parts in software application and let them run simultaneously based on hardware parallelism. For example, in the FFT-based polynomial multiplication, the DFT of the input polynomials are independent, hence, they can be computed simultaneously. Another example is the (standard) Chinese remaindering algorithm, where the computations w.r.t. each modulo can be performed simultaneously.

In fact, under the Linux environment, we directly use the native Posix Thread Library to conduct parallel programming, since AXIOM's compiler is not able to provide this kind of optimization. This part is relatively simple but we have not completed yet extensive benchmarks. However, we have observed that the parallelized version of the FFT-based multiplication is 7 to 10 percent faster than the

non-parallelized version on our dual CPU machine as shown by Figure 5.4.

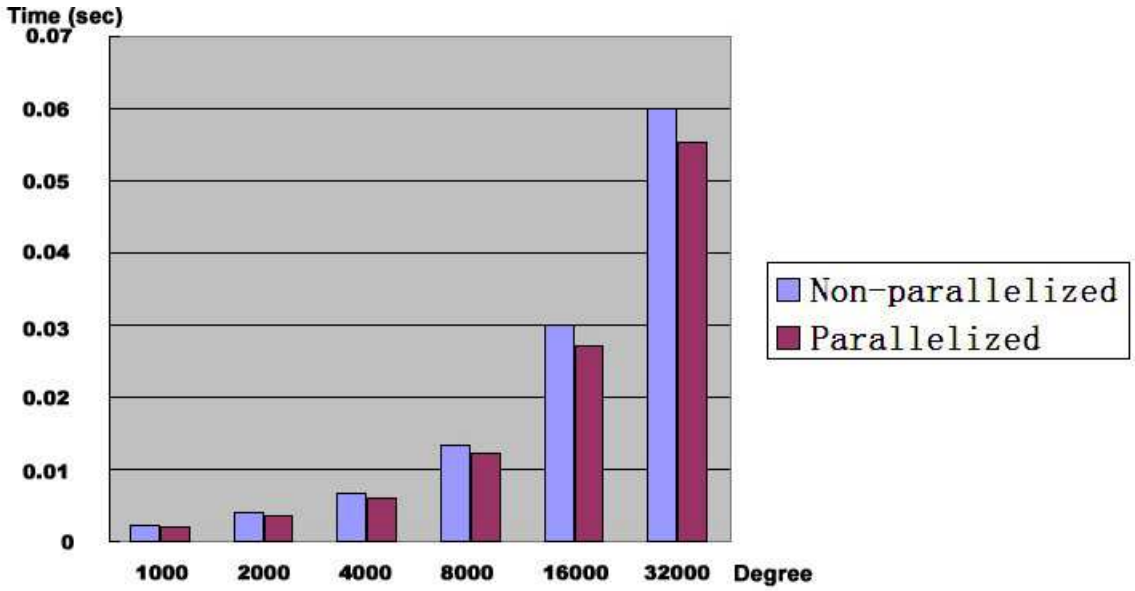

Figure 5.4: Parallelized vs. non-parallelized version of FFT-based univariate polynomial multiplicaiton over $\mathbb{Z}/p\mathbb{Z}$.

Chapter 6

FFT-based univariate polynomial multiplication

6.1 Introduction

The Fourier transform converts a signal representation from the time-domain to the frequency domain. The Discrete Fourier Transform (DFT) is a Fourier transform which operates on a discrete function instead of a continuous function. The Fast Fourier Transform (FFT) is a fast algorithm for calculating the DFT of a function. This algorithm was essentially known to Gauss and was reinvented by Cooley and Turkey in 1965. See [17] for more historical details.

In symbolic computations, the FFT algorithm has many applications [13]. The most famous one is the fast multiplication of polynomials. Even if the principles of these calculations are quite simple, their practical implementations are still an active area of investigation. This chapter and the next one describe our contribution to this area. Respectively, these chapters treat the univariate and the multivariate cases. Our experimental results show that our implementation of the FFT-based symbolic polynomial multiplication competes with, and often outperforms, the best known comparable packages.

6.2 The FFT-based univariate multiplication

The principles of the FFT-based univariate polynomial multiplication are sketched on Figure 6.1.

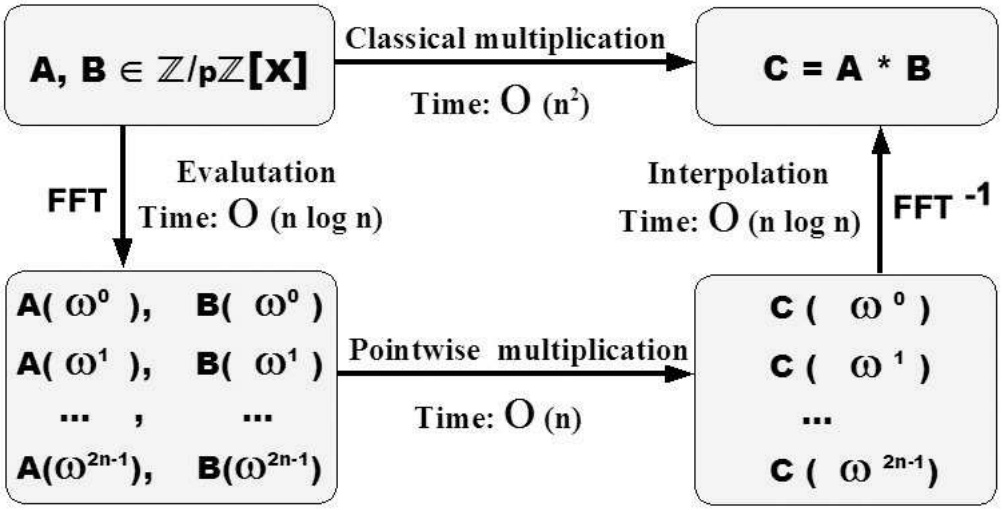

Figure 6.1: FFT-based univariate polynomial multiplication over $\mathbb{Z}/p\mathbb{Z}$.

In this chart, we consider two polynomials $f = \sum_{k=0}^{k=n-1} a_k x^k$ and $g = \sum_{k=0}^{k=n-1} b_k x^k$ over some field $\mathbb{K}$. We do not need to assume that they have the same degree; if they do not have the same degree, we add a "zero leading coefficient" to the one of smaller degree. We want to compute the product $fg = \sum_{k=0}^{k=2n-2} c_k$. The classical algorithm would compute the coefficient c_k of fg by

$$c_k = \sum_{i=0}^{i=k} a_i b_{k-i} \tag{6.1}$$

for $k = 0, \ldots, 2n-2$, amounting to $O(n^2)$ operations in $\mathbb{K}$.

The first idea in the FFT-based univariate polynomial multiplication is as follows: if the values of f and g are known at $2n-1$ different points of $\mathbb{K}$, say $x_0, \ldots, x_{2n-2}$ then we can obtain fg by computing $f(x_0)g(x_0), \ldots, f(x_{2n-2})g(x_{2n-2})$ amounting to $O(n)$ operations in $\mathbb{K}$.

The second idea is to use points $x_0, \ldots, x_{2n-1}$ in $\mathbb{K}$ such that

(i) evaluating f and g at these points can be done in nearly linear time cost, such as $O(n\log(n))$,

(*ii*) interpolating the values $f(x_0)g(x_0), \ldots, f(x_{2n-2})g(x_{2n-2})$ can be done in nearly linear time, that is $O(n\log(n))$ again.

Such points $x_0, \ldots, x_{2n-2}$ do not always exist in $\mathbb{K}$. However, there are techniques to overcome this limitation (essentially by considering a field extension of $\mathbb{K}$ where the desired points can be found). In the end, this leads to an algorithm for FFT-based univariate polynomial multiplication which runs in $O(n\log(n)\log(\log(n)))$ operations in $\mathbb{K}$ [4]. This is the best known algorithm for arbitrary $\mathbb{K}$. Recall that the Karatsuba's algorithm runs in $O(n^{\log(3)/\log(2)})$. See [17] for more details.

In this thesis, we restrict ourselves to the case where we can find points $x_0, \ldots, x_{2n-2}$ in $\mathbb{K}$ satisfying the above (*i*) and (*ii*). Most finite fields possess such points for n small enough. (Obviously n must be at most equal to the cardinality of the field.) More precisely, for $n, p > 1$, where p is a prime, the finite field $\mathbb{Z}/p\mathbb{Z}$ has a primitive m-th root of unity if and only if m divides $p-1$. (Recall that $\omega \in \mathbb{K}$ is a primitive m-th root of unity of the field $\mathbb{K}$ if and only if $\omega^m = 1$ and $\omega^k \neq 1$ for $0 < k < m$.) If $\mathbb{Z}/p\mathbb{Z}$ has a primitive m-th root of unity ω,

- then we use $x_k = \omega^k$ for $k = 0, \ldots, 2n-2$,
- Step (*i*) is the FFT of f and g at ω (to be detailed in the next section),
- Step (*ii*) is the FFT of $\frac{fg}{n}$ at ω^{-1}.

Again, we refer to [17] for more details.

6.3 Efficient algorithm for the FFT

The theoretical complexity of the FFT-based univariate multiplication is nearly optimal. However, in order to obtain an efficient code, one needs to carefully implement the FFT calculations. The FFT algorithm is a divide-and-conquer algorithm. Its presentation in [17] is recursive and not suitable for producing an optimized code. In this section, we reproduce the iterative presentation of [28] (which appears in other places) and discuss an efficient implementation of the FFT. This leads us to

change the notation a little bit, w.r.t. the previous section. Our input polynomial becomes

$$A(x) = \Sigma_{0 \leq i < n}\, a_i\, x^i \tag{6.2}$$

with degree at most $n/2 - 1$. Let $\omega \in \mathbb{K}$ be a primitive n-th root of unity. We assume that $n = 2^r$ is a power of 2. We aim at computing the FFT of A at ω, that is to evaluate A at the points ω^k for $k = 0, \ldots, n-1$. We define

$$\begin{aligned} A^{[0]}(x) &= a_0 + a_2 x + a_4 x^2 + \cdots + a_{n-2} x^{n/2-1} \\ A^{[1]}(x) &= a_1 + a_3 x + a_5 x^2 + \cdots + a_{n-1} x^{n/2-1} \end{aligned} \tag{6.3}$$

such that we have

$$A(x) \;=\; A^{[0]}(x^2) + x\, A^{[1]}(x^2). \tag{6.4}$$

Hence evaluating A at $1, \omega, \omega^2, \ldots, \omega^{n-1}$ is reduced to evaluating $A^{[0]}$ and $A^{[1]}$ at $1, \omega^2, \omega^4, \ldots, \omega^{2n-2}$. But this second sequence of powers of ω consists of two identical sequences of length $n/2$. Indeed, for every integer k we have

$$(\omega^{k+n/2})^2 \;=\; (\omega^k)^2 \tag{6.5}$$

Since $\omega^k(1 - \omega^{n/2}) \neq 0$ holds, we obtain

$$\omega^{k+n/2} \;=\; -\omega^k \tag{6.6}$$

Hence we only need to evaluate $A^{[0]}$ and $A^{[1]}$ at

$$1, \omega^2, \omega^4, \ldots, \omega^{n-2}. \tag{6.7}$$

This provides us directly with the values of A at

$$1, \omega, \omega^2, \ldots, \omega^{n/2-1} \tag{6.8}$$

and the values of A at

$$\omega^{n/2}, \omega^{n/2+1}, \ldots, \omega^{n-1} \tag{6.9}$$

by using the fact that numbers are respectively equal to

$$-1, -\omega, -\omega^2, \ldots, -\omega^{n/2-1}. \tag{6.10}$$

To summarize, for $0 \leq i \leq n/2 - 1$ we have

$$\begin{aligned} A(\omega^i) &= A^{[0]}(\omega^{2i}) + \omega^i A^{[1]}(\omega^{2i}) \\ A(\omega^{i+n/2}) &= A^{[0]}(\omega^{2i}) - \omega^i A^{[1]}(\omega^{2i}) \end{aligned} \tag{6.11}$$

Algorithm 1

Input: $n = 2^r$, $A = [a_0, a_1, \ldots, a^{n/2-1}, \ldots, a^{n-1}]$ an array for the coefficient of the polynomial, $\Omega = [1, \omega, \ldots, \omega^{n/2-1}, \ldots, \omega^{n-1}]$ an array of the powers of a primitive n-th root of unity ω.

Output: An array $y = [y_0, \ldots, y_{n-1}]$ such that y_i is $A(\omega^i)$ by calling $DFT(n, A, \Omega)$.

$DFT(n, A, \Omega) ==$
  $step := \#(\Omega)/n$
  if $n = 1$ **return** $[f_0]$
  $A^{[0]} := [a_0, a_2, \ldots, a_{n-2}]$
  $A^{[1]} := [a_1, a_3, \ldots, a_{n-1}]$
  $y^{[0]} := DFT(n/2, A^{[0]}, \Omega)$
  $y^{[1]} := DFT(n/2, A^{[1]}, \Omega)$
  for $k := 0 \cdots n/2 - 1$ **repeat**
    $s :=$ step k
    $t := \Omega_s \;\; y_k^{[1]}$
    $y_k := y_k^{[0]} + t$
    $y_{k+n/2} := y_k^{[0]} - t$
  return y

The validity of Algorithm 1 follows from the two following observations.

- Formulas (6.11).
- If ω is a primitive 2^s-root of unity, with $s > 1$, then ω^2 is a primitive 2^{s-1}-root of unity.

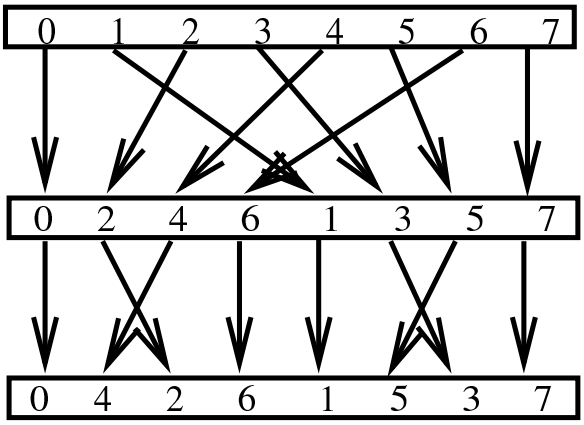

Figure 6.2: Recursive calls for $n = 8$.

Observe that the recursive calls of $DFT(n, A, \Omega)$ define an ordering of the coefficients of A shown on Figure 6.2 for $n = 8$. Let us call this ordering the *DFT ordering* of A.

Observe that if the input array A is sorted w.r.t. this DFT ordering, one can describe easily the recursive calls with a binary tree. If this tree is traversed bottom-up, from left to right, we are led to **an iterative algorithm working in place!** described in Algorithm 2.

Algorithm 2

Input: $n = 2^r$, A the array for the coefficient of the polynomial sorted by the *DFT ordering*. $\Omega = [1, \omega, \ldots, \omega^{n/2-1}, \ldots, \omega^{n-1}]$ an array of the powers of a primitive n-th root of unity ω.

Output: An array $y = [y_0, \ldots, y_{n-1}]$ such that y_i is $A(\omega^i)$ by calling $DFT(n, A, \Omega)$.

```
DFT(n, A, Ω) ==
  r := log2(n)
  for s := 1 ··· r repeat
  # Traversing the tree, bottom-up.
    m := 2^s
    for k := 0 ··· n − 1 by m repeat
    # for each internal node from left to right
      for j = 0 ··· m/2 − 1 repeat
      # combine its two children
        t := Ω_jm  A[k + j + m/2]
        u := A[k + j]
        A[k + j] := u + t
        A[k + j + m/2] := u − t
  return A
```

We conclude this section with an observation regarding code optimization. For $n = 8$, assuming that the input array A is

$$| a_0 | a_4 | a_2 | a_6 | a_1 | a_5 | a_3 | a_7 |,$$

we obtain the desired result in A using the following straight-line program:

1. $a := a_0$
2. $b := a_1$
3. $a_0 := a + b$
4. $a_1 := a - b$

5. $a := a_2$
6. $b := a_3$
7. $a_2 := a + b$
8. $a_3 := a - b$
9. $a := a_4$
10. $b := a_5$
11. $a_4 := a + b$
12. $a_5 := a - b$
13. $a := a_6$
14. $b := a_7$
15. $a_6 := a + b$
16. $a_7 := a - b$
17. $a := a_0$
18. $b := a_1$
19. $c := a_2$
20. $d := a_3$
21. $a_0 := a + c$
22. $a_2 := a - c$
23. $a_1 := b + \omega d$
24. $a_3 := b - \omega d$
25. $a := a_4$
26. $b := a_5$
27. $c := a_6$

28. $d := a_7$
29. $a_0 := a + c$
30. $a_2 := a - c$
31. $a_1 := b + \omega d$
32. $a_3 := b - \omega d$
33. $a := a_0$
34. $b := a_1$
35. $c := a_2$
36. $d := a_3$
37. $e := a_4$
38. $f := a_5$
39. $g := a_6$
40. $h := a_7$
41. $a_0 := a + e$
42. $a_1 := b + \omega f$
43. $a_2 := c + \omega^2 f$
44. $a_3 := d + \omega^3 h$
45. $a_4 := a + e$
46. $a_5 := b - \omega f$
47. $a_6 := c - \omega^2 f$
48. $a_7 := d - \omega^3 h$

More generally, for small values of n, we can implement Algorithm 2 by a straight-line program. This appears in practice to be an important optimization.

6.4 Computing primitive roots of unity

A preliminary step of the FFT-based multiplication is to find primitive n-th roots of unity. We explain in this section how to compute such numbers in $\mathbb{K} = \mathbb{Z}/p\mathbb{Z}$. Of course, we assume that n divides $p-1$. Hence, there exists an integer q such that $p = qn + 1$ holds. According to little Fermat's theorem, for all $\alpha \in \mathbb{K}$, with $\alpha \neq 0$ and $\alpha \neq 1$, we have

$$\alpha^{p-1} = 1. \tag{6.12}$$

Therefore, we obtain

$$\alpha^{qn} = 1. \tag{6.13}$$

It follows that α^q is a candidate for being a primitive n-th root of unity. If α^q is not a primitive n-root root of unity then:

- the sequence of the α^{qk} for $k = 0, \ldots, n-1$ is periodic, with a period dividing n, and
- in particular $\alpha^{qn/2} = 1$ holds.

Since $\alpha^{qn/2}$ equals either 1 or -1, we have the following.

Proposition 1 *Let $p > 2$ be a prime number and let n, q be integers such that n is a power of 2 $p = nq + 1$. Let $\alpha \in \mathbb{Z}/p\mathbb{Z}$ different from 0 and 1. Then, α^q is a primitive n-th root of unity if and only if $\alpha^{qn/2} = -1$ holds.*

This proposition is very easy to implement and provides an efficient way to compute primitive roots of unity in practice. It avoids the construction of tables of primitive roots of unity, which was a usual approach, see [28]. The code below shows our SPAD code implementing a probabilistic algorithm based on this proposition.

```
1     getNthRoots (nth: Integer)==
2          theroots:=ARRAYFIX(nth)$Lisp
3          k:PositiveInteger:=((p-1) quo nth)
4          n2:PositiveInteger:=(nth quo 2)
```

```
5          root:R
6          repeat
7                root:=random()$R  --Z/pZ
8                if not((root=0) or (root=1)) then
9                         xroot:R:=(root**k)
10                        not (one? (xroot**n2))=> break
11         GETROOTS(theroots, p, xroot, nth)$Lisp
12         [theroots, xroots]$REC
```

6.5 Implementation of the small prime case

Following the discussion of Chapter 4, our implementation of the FFT-based univariate polynomial multiplication distinguishes two cases:

- the small prime case, presented in this section,
- the big prime case, discussed in the next one.

We implement Algorithm 2 using the strategies presented in Chapter 5, such as reducing memory traffic. In addition, we use the following techniques.

- As illustrated above, for n small, we "pre-compute strait line programs" and use them instead of a direct implementation of Algorithm 2 (which contains nested loops). Experiments show that there is a "cut-off" point for this strategy, that is, a value n_0 such that for $n \geq n_0$ this strategy should not be used, since it generates too large strait line programs.
- We pre-compute the DFT ordering for the coefficient array input of Algorithm 2.
- We unroll some inner loops and inline some small and frequently used functions.

- We write both generic and MMX/SSE2 ASSEMBLY code for the crucial modular arithmetic operations:
  - $(a, b, p) \longmapsto a + b \mod p$,
  - $(a, b, p) \longmapsto a - b \mod p$,
  - $(a, b, p) \longmapsto ab \mod p$.

6.6 Implementation of the big prime case

Two strategies are possible in this case.

Using GMP. One can directly implement Algorithm 2 by adapting the code of the small prime case to the big prime case. This implies using big integer (= multiple precision) arithmetic from the `GMP library` for the coefficients of the polynomials, as described in Chapter 4. In some sense, this is a bit contradictory with the spirit of modular computations.

Using the CRA approach. Alternatively, one can proceed as follows, as suggested by several authors [17, 35].

1. View the input polynomials $f, g \in \mathbb{Z}/p\mathbb{Z}[x]$ as polynomials over $\mathbb{Z}$. Let n be the smallest power of 2 greater than the degree of the product fg.
2. Using Formula (6.1), obtain an upper bound C for the coefficients of fg (viewed over $\mathbb{Z}$).
3. Choose small primes $2 < p_1 < \ldots < p_s$ such that their product m exceeds $2C$ and such that for all $i = 1, \ldots, s$ the field $\mathbb{Z}/p_i\mathbb{Z}$ admits primitive n-th root of unity. (In practice, one can always find such primes, see [17].)
4. Compute $fg \in \mathbb{Z}/p_i\mathbb{Z}[x]$ for all $i = 1, \ldots, s$ and combine the results termwise, with the Chinese Remaindering Algorithm, such that the coefficients of fg are in the symmetric range $-m \ldots m$. This gives the product fg in $\mathbb{Z}[x]$.

5. Reduce each coefficient of this product modulo p.

Figure 6.3 shows an experimental comparison between these two approaches.

We also put an intensive effort on the double precision big prime case. We rewrote some GMP low-level functions and implement high-performance Assembly versions of CRA for this special case.

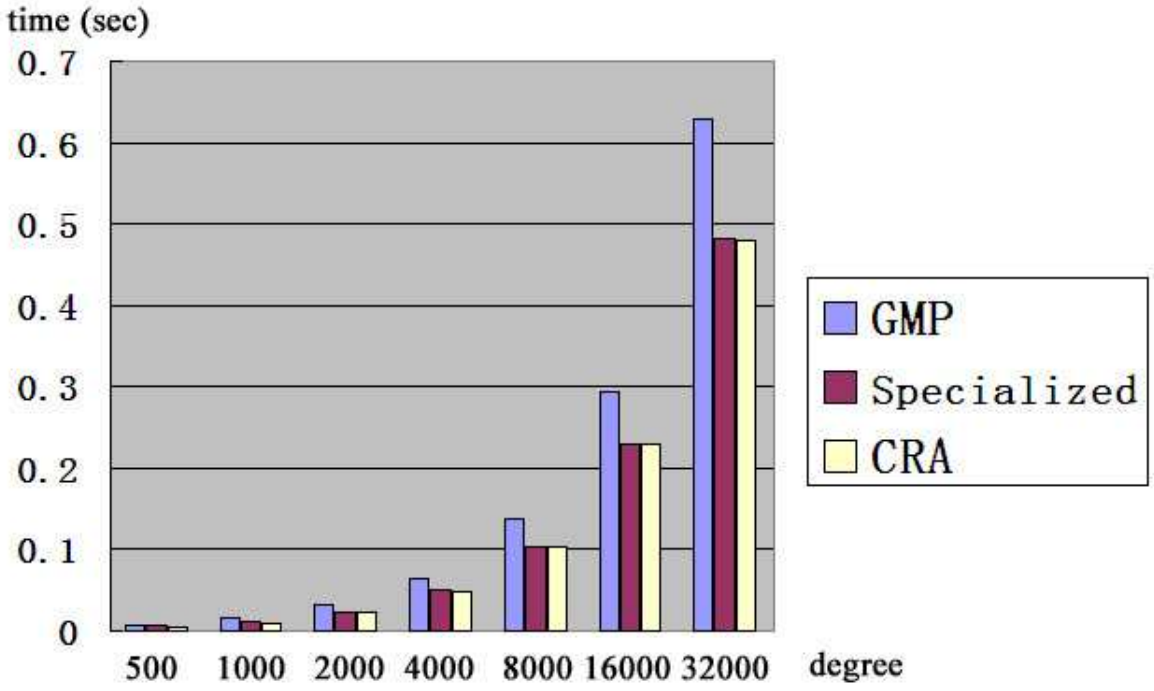

Figure 6.3: FFT-based univariate polynomial multiplicaiton over $\mathbb{Z}/p\mathbb{Z}$, GMP functions vs our specialized double percision integer functions and CRA-based functions.

Figure 6.3 shows that our specialized double precision big prime functions and CRA-based approaches are faster than the generic GMP functions. The CRA recombination part spends 0.06 to 0.07 percent of the time in the whole FFT algorithm which is negligible.

6.7 The FFT in the NTL library

NTL is a high-performance, portable C++ library providing data structures and algorithms for calculating with signed, arbitrary length integers, and for manipulating vectors, matrices, and polynomials over the integers and over finite fields. [34] The principal author of NTL is Dr. Victor Shoup.

The FFT-based univariate polynomial multiplication in NTL is one of best known implementations which is based on the Schoenhage-Strassen approach and uses small prime numbers during computation. The Schoenhage-Strassen's multiplication algorithm works for any arbitraryrRing. This requires constructing "virtual" roots of unity.

The Figure 6.4 and the Figure 6.5 are two benchmarks. In these two benchmarks, we used dense univariate represenation for every input polynomial. These input polynomials in AXIOM and MAGMA are randomly generated. And every term in a polynomial is non-zero.

Our implementation is faster than NTL over small prime case, however, slower than NTL over big primes (Our implementation is faster than MAGMA and other known CAS in both small and big primes). The main reason we believe is that NTL re-arranges the computations in a more "cache-friendly" way than our implementation. However, we are implementing Truncated Fourier Transform(TFT) [21] and developing cache-friendly code for this TFT by using the strategies from [23]. We expect that our new TFT implementation will be faster than NTL's FFT.

6.8 Benchmarks

In the Figure 6.4 input polynomials are unvariate over prime number 65535. In the Figure 6.5 input polynomials are unvariate over big prime numbers

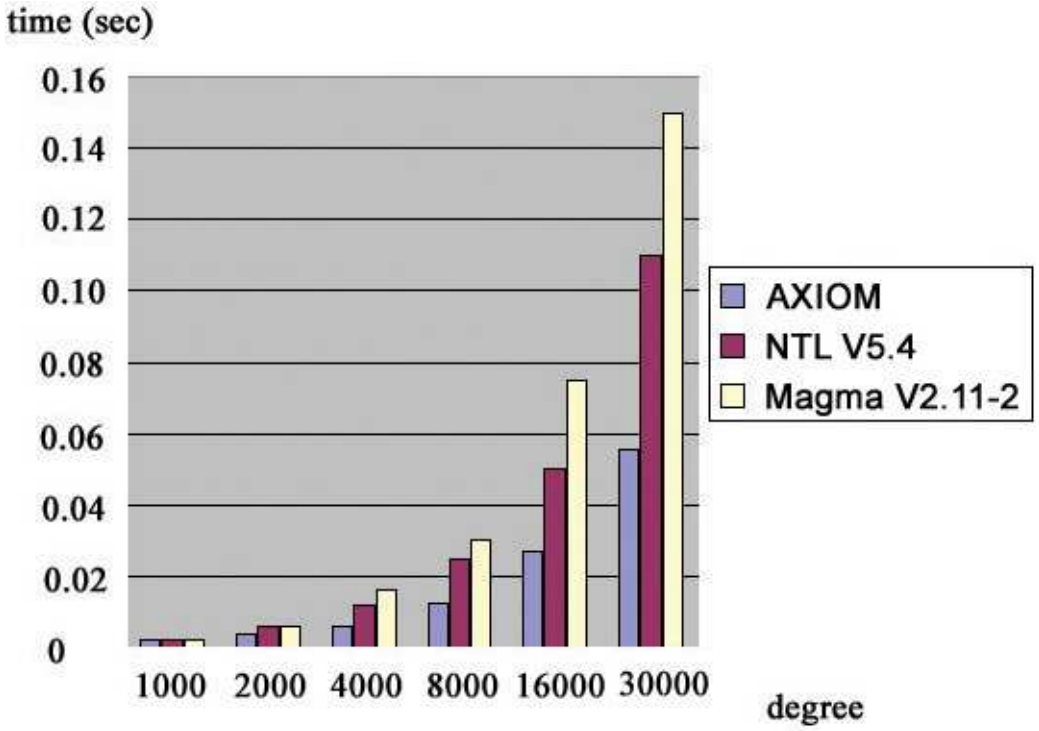

Figure 6.4: Benchmark of FFT with small primes.

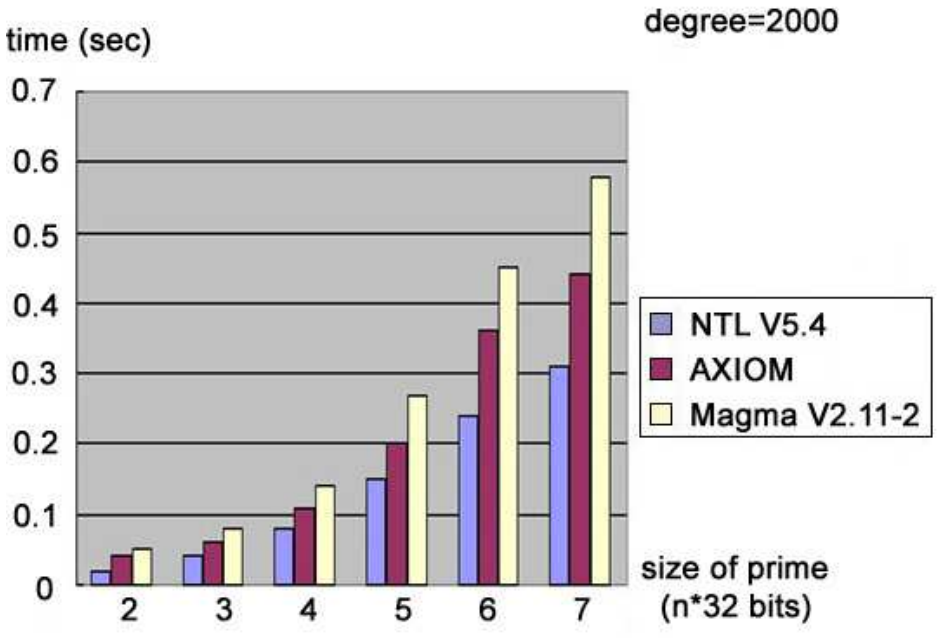

Figure 6.5: Benchmark of FFT with big primes.

Chapter 7

FFT-based multivariate polynomial multiplication

7.1 Introduction

In the previous chapter, we discussed FFT-based univariate polynomial multiplication. This is a very important issue since most fast algorithms for polynomial arithmetic rely directly or indirectly on it. In this chapter, we discuss multivariate polynomial multiplication. We reduce this to the univariate case through Kronecker's substitution [17]. We restrict again to coefficient fields of the form $\mathbb{Z}/p\mathbb{Z}$, where p is a prime, since one of our main objectives is to provide highly efficient support for implementing modular methods. Using our code for the FFT-based univariate polynomial multiplication, described in Chapter 6, we obtain a very efficient implementation of the multivariate polynomial multiplication, which outperforms the computer algebra system MAGMA. This chapter is a joint work with my supervisor, Marc Moreno Maza and Éric Schost (École Polytechnique, France). It is based on a preprint article [26].

7.2 Kronecker's substitution

Let $\mathbb{A}$ be a commutative ring with one. Let $X_1 < X_2 < \cdots < X_n$ be n ordered variables and let $\alpha_1, \alpha_2, \ldots, \alpha_n$ be n positive integers with $\alpha_1 = 1$. We consider the ideal $\mathcal{I}$ of $\mathbb{A}[X_1, X_2, \ldots, X_n]$ generated by $X_2 - X_1^{\alpha_2}, X_3 - X_1^{\alpha_3}, \ldots, X_n - X_1^{\alpha_n}$. Define $\alpha = (\alpha_1, \alpha_2, \alpha_3, \ldots, \alpha_n)$. Let Ψ_α be the canonical map from $\mathbb{A}[X_1, X_2, \ldots, X_n]$ to $\mathbb{A}[X_1, X_2, \ldots, X_n]/\mathcal{I}$, which substitutes the variables X_2, X_3, ..., X_n with $X_1^{\alpha_2}$, $X_1^{\alpha_3}$, ..., $X_1^{\alpha_n}$ respectively. We call it the Kronecker map of α. This map transforms a multivariate polynomial of $\mathbb{A}[X_1, X_2, \ldots, X_n]$ into a univariate polynomial of $\mathbb{A}[X_1]$. It has the following immediate property.

Proposition 2 *The map Ψ_α is a ring-homomorphism (see [18] p. 153 for this term). In particular, for all $a, b \in \mathbb{A}[X_1, X_2, \ldots, X_n]$ we have*

$$\Psi_\alpha(ab) = \Psi_\alpha(a)\Psi_\alpha(b). \tag{7.1}$$

Therefore, if the product $\Psi_\alpha(a)\Psi_\alpha(b)$ has only one pre-image by Ψ_α, one can compute the product of the multivariate polynomials a and b via the product of the univariate polynomials $\Psi_\alpha(a)$ and $\Psi_\alpha(b)$. This is advantageous, when one has at hand a fast univariate multiplication. In order to study the pre-images of $\Psi_\alpha(a)\Psi_\alpha(b)$ we introduce additional material.

Let $d_1, d_2, \ldots, d_n$ be non-negative integers. We write $\mathbf{d} = (d_1, d_2, \ldots, d_n)$ and we denote by Δ_n the set of the n-tuples $\mathbf{e} = (e_1, e_2, \ldots, e_n)$ of non-negative integers such that $e_i \leq d_i$ for all $i = 1, \ldots, n$. We define

$$\delta_n = d_1 + \alpha_2 d_2 + \cdots + \alpha_n d_n \quad \text{and} \quad \delta_0 = 0, \tag{7.2}$$

and we consider the map $\psi_{\mathbf{d}}$ defined by

$$\begin{array}{rcl} \psi_{\mathbf{d}} : \Delta_n & \longrightarrow & [0, \delta_n] \\ (e_1, e_2, \ldots, e_n) & \longmapsto & e_1 + \alpha_2 e_2 + \cdots + \alpha_n e_n \end{array} \tag{7.3}$$

that we call the *packing exponent map*.

Proposition 3 *The* packing exponent map $\psi_{\mathbf{d}}$ *is an one-to-one map if the following relations holds:*

$$\begin{aligned} \alpha_2 &= 1 + d_1, \\ \alpha_3 &= 1 + d_1 + \alpha_2 d_2, \\ \vdots &\quad \vdots \\ \alpha_n &= 1 + \sum_{i=1}^{i=n-1} \alpha_i d_i, \end{aligned}$$

that we call the packing relations.

PROOF. We proceed by induction on $n \geq 1$. For $n = 1$, we have $\delta_1 = d_1$ and $\psi_{\mathbf{d}}(e_1) = e_1$ for all $0 \leq e_1 \leq d_1$. Thus, the packing exponent map $\psi_{\mathbf{d}}$ is clearly an one-to-one map in this case. Since the packing relations trivially hold for $n = 1$, the property is proved in this case.

We consider now $n > 1$ and we assume that the property holds for $n - 1$. We look for necessary and sufficient conditions for $\psi_{\mathbf{d}}$ to be an one-to-one map. We observe that the partial function

$$\psi_{(d_1,\ldots,d_{n-1})} : \begin{array}{ccc} \Delta_{n-1} & \longrightarrow & [0, \delta_{n-1}] \\ (e_1, e_2, \ldots, e_{n-1}) & \longmapsto & \psi_{\mathbf{d}}(e_1, e_2, \ldots, e_{n-1}, 0) \end{array} \tag{7.4}$$

of ψ_d needs to be an one-to-one map for ψ_d to be an one-to-one map. Therefore, by induction hypothesis, we can assume that the following relations hold

$$\begin{aligned} \alpha_2 &= 1 + d_1, \\ \alpha_3 &= 1 + d_1 + \alpha_2 d_2, \\ \vdots &\quad \vdots \\ \alpha_{n-1} &= 1 + \sum_{i=1}^{i=n-2} \alpha_i d_i. \end{aligned}$$

Observe that the last relation writes

$$\alpha_{n-1} = 1 + \delta_{n-2}. \tag{7.5}$$

We consider now $f \in [0, \delta_n]$. Let q and r be the quotient and the remainder r of the Euclidean division of f by α_n. Hence, we have

$$f = q\alpha_n + r \quad \text{and} \quad 0 \leq r < \alpha_n. \tag{7.6}$$

Moreover, the couple (q, r) is unique with these properties. Assume that $\alpha_n = 1 + \delta_{n-1}$ holds then f has a unique pre-image in Δ_n by ψ_d^{-1} which is

$$\psi_d^{-1}(f) = (\psi_{(d_1,\ldots,d_{n-1})}^{-1}(r), q). \tag{7.7}$$

If $\alpha_n > 1 + \delta_{n-1}$ holds, then $f = 1 + \delta_{n-1}$ has no pre-images in Δ_n by ψ_d^{-1}. If $\alpha_n < 1 + \delta_{n-1}$ holds, then $f = \alpha_n$ has two pre-images in Δ_n, namely

$$(0, \ldots, 0, 1) \text{ and } \psi_{(d_1,\ldots,d_{n-1})}^{-1}(\alpha_n). \tag{7.8}$$

Finally, the map $\psi_{\mathbf{d}}$ is one-to-one if and only if the packing relations hold. □

Proposition 4 *Let* $\mathbf{e} = (e_1, e_2, \ldots, e_n)$ *be in* Δ_n *and* $\mathbf{X} = X_1^{e_1} X_2^{e_2} \cdots X_n^{e_n}$ *be a monomial of* $\mathbb{A}[X_1, X_2, \ldots, X_n]$. *We have*

$$\Psi_\alpha(\mathbf{X}) = X_1^{\psi_{\mathbf{d}}(\mathbf{e})}. \tag{7.9}$$

Moreover, for all $f = \sum_{\mathbf{X} \in S} c_{\mathbf{X}} \mathbf{X}$

$$\Psi_\alpha(f) = \sum\nolimits_{\mathbf{X} \in S} c_{\mathbf{X}} \Psi_\alpha(\mathbf{X}) \tag{7.10}$$

where S *is the support of* f, *that is the set of the monomials occurring in* p.

Proof. Relation (7.9) follows easily from the definition of Ψ_α. Relation (7.10) follows from Proposition 2. □

We denote by $\mathbb{A}[\Delta_n]$ the set of the polynomials $p \in \mathbb{A}[X_1, X_2, \ldots, X_n]$ such that for every $\mathbf{X} = X_1^{e_1} X_2^{e_2} \cdots X_n^{e_n}$ in the support of p we have $(e_1, e_2, \ldots, e_n) \in \Delta_n$. The set $\mathbb{A}[\Delta_n]$ is not closed under multiplication, obviously. Hence it is only a $\mathbb{A}$-module (see [12] p. 317). The same remark holds for the set $\mathbb{A}[\delta_n]$ of univariate polynomials over $\mathbb{A}$ with degree less than or equal to δ_n.

Proposition 5 *Assume that packing relations hold. Then, the map* Ψ_α *defines an* A*-module isomorphism between* $\mathbb{A}[\Delta_n]$ *and* $\mathbb{A}[\delta_n]$.

Proof. The fact that Ψ_α defines an A-module homomorphism from $\mathbb{A}[\Delta_n]$ to $\mathbb{A}[\delta_n]$ follows immediately from Proposition 2. The fact that, this A-module

homomorphism is surjective, is also clear. Indeed, it follows from Proposition 3 that every monomial of $\mathbb{A}[\delta_n]$ has a (unique) monomial pre-image in $\mathbb{A}[\Delta_n]$. Now, let $f, g \in \mathbb{A}[\Delta_n]$. Assume that $\Psi_\alpha(f) = \Psi_\alpha(g)$ holds. Since Ψ_α is an A-module homomorphism, we have $\Psi_\alpha(f - g) = 0$. Observe that the image of every non-zero polynomial of $\mathbb{A}[\Delta_n]$ is non-zero. Hence, we must have $f = g$. □

7.3 Fast multivariate multiplication

We use the same notations and hypothesis as in Section 7.2. Although the restriction of the map Ψ_α to $\mathbb{A}[\Delta_n]$ is not a ring isomorphism, it can be used for multiplying multivariate polynomials as follows.

Let $f, g \in \mathbb{A}[X_1, X_2, \ldots, X_n]$ and let p be their product. For all $1 \leq i \leq n$ we choose

$$d_i = \deg(f, X_i) + \deg(g, X_i), \tag{7.11}$$

that is the sum of the partial degrees of f and g w.r.t. X_i. Observe that for all $1 \leq i \leq n$ we have

$$\deg(p, X_i) \leq \deg(f, X_i) + \deg(g, X_i). \tag{7.12}$$

It follows that the three polynomials f, g, p belong to $\mathbb{A}[\Delta_n]$. Moreover, from Proposition 2, we have

$$\Psi_\alpha(p) = \Psi_\alpha(f)\Psi_\alpha(g). \tag{7.13}$$

Therefore, we can compute p using the following simple algorithm:

Algorithm 3

Input: $f, g \in \mathbb{A}[\Delta_n]$ such that $fg \in \mathbb{A}[\Delta_n]$ holds.

Output: fg

1 $u_f := \Psi_\alpha(f)$
2 $u_g := \Psi_\alpha(g)$
3 $u_{fg} := u_f u_g$
4 $p := \Psi_\alpha^{-1}(u_{fg})$
5 **return** p

Let us assume that we have at hand a quasi-linear algorithm for multiplying in $\mathbb{A}[X_1]$, that is an algorithm such that the product of two polynomials of degree less that k can be computed in $O(k^{1+\epsilon})$ operations in $\mathbb{A}$, for all $\epsilon > 0$. Such algorithm exists over any ring $\mathbb{A}$ [4]. It follows that step **3** of the above algorithm can be performed in $O({\delta_n}^{1+\epsilon})$ for every $\epsilon > 0$. Therefore, we have:

Proposition 6 *For every $\epsilon > 0$, Algorithm 3 runs in $O((d_1 + n) \cdots (d_n + 1))^{1+\epsilon})$ operations in $\mathbb{A}$.*

PROOF. First, we observe that each one of the Steps **1**, **2**, **4** and **5** performs in $O(\delta_n)$ operations in $\mathbb{A}$. Hence, the cost of Algorithm 3 is that of its third step. Next, we observe that for $n > 1$ we have

$$\begin{aligned} \delta_n &= \alpha_1 d_1 + \alpha_2 d_2 + \cdots + \alpha_n d_n \\ &= \delta_{n-1} + \alpha_n d_n \\ &= \delta_{n-1} + (\delta_{n-1} + 1) d_n \\ &= (d_n + 1)\delta_{n-1} + d_n. \end{aligned}$$

Now, we prove by induction on $n \geq 1$ that the relation below holds

$$d_1 \cdots d_n \leq \delta_n \leq (d_1 + n) \cdots (d_n + 1). \tag{7.14}$$

This is clearly true for $n = 1$. Let $n > 1$ and assume that the relation holds for

$n-1$. Hence, we have

$$d_1 \cdots d_{n-1} \leq \delta_{n-1} \leq ((d_1 + n - 1) \cdots (d_{n-1} + 1).$$

We deduce

$$(d_n + 1)d_1 \cdots d_{n-1} + d_n \leq \delta_n \leq (d_1 + n - 1) \cdots (d_{n-1} + 1)(d_n + 1) + d_n.$$

We obtain:

$$d_n d_1 \cdots d_{n-1} \leq \delta_n \leq (d_1 + n) \cdots (d_{n-1} + 1)(d_n + 1),$$

as claimed. Therefore, Step **3** performs $O(((d_1 + n) \cdots (d_n + 1))^{1+\epsilon})$ operations in $\mathbb{A}$. □

Is δ_n independent of the variable ordering $X_1 < X_2 < \cdots < X_n$? The answer is yes! Indeed, the following result shows that δ_n is a symmetric function of $(d_1, \ldots, d_n)$.

Proposition 7 *For $n \geq 1$ we have*

$$\delta_n = \sum_{k=1}^{k=n} \sum_{1 \leq i_1 < i_2 < \cdots < i_k \leq n} d_{i_1} \cdots d_{i_k}. \tag{7.15}$$

Proof. This follows by induction from $\delta_n = (d_n + 1)\delta_{n-1} + d_n$. □

7.4 Benchmarks

We have realized benchmarks between the computer algebra system MAGMA and our code:

Figures	n	p	Variable
7.1	2	5767169	δ_n
7.2	3	23068673	δ_n
7.3	2	18446744073692774401	δ_n
7.4	3	18446744073692774401	δ_n

where d here stand for the maximum of the partial degrees $d_1, \ldots, d_n$. δ represents the number of expanding terms after Kronecker substitution.

In these benchmarks, we used dense representations for the randomly generated input polynomials. Every term in each input polynomial is non-zero. We have

checked every input polynomial in both AXIOM and MAGMA to make sure that we are comparing the same thing.

According the benchmark results, our FFT-based multivariate polynomial multiplication over $\mathbb{Z}/p\mathbb{Z}$ outperforms MAGMA's counterpart in any case. From the Figure 7.1 we can observe that when MAGMA is in the "classical multiplication" stage, our FFT-based implementation is already faster. From Figures 7.2, 7.3, and 7.4 we can observe that both our FFT and MAGMA's FFT are showing staircase-like curves. This is the characteristic of FFT algorithm. And our implementation is 2 times faster than MAGMA's implementation.

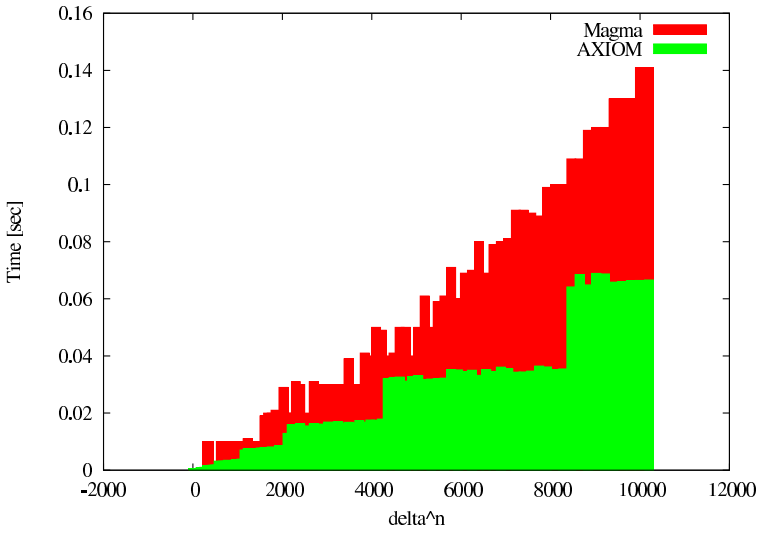

Figure 7.1: FFT-based bivariate polynomial multiplication modulo 5767169.

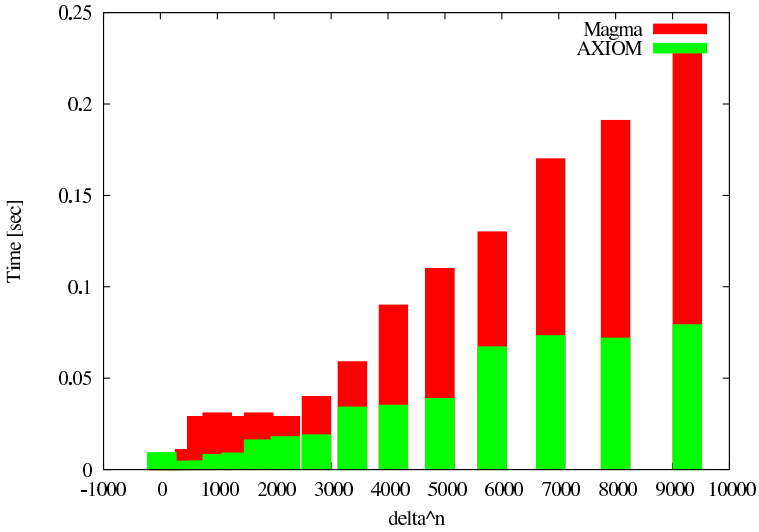

Figure 7.2: FFT-based trivariate polynomial multiplication modulo 23068673.

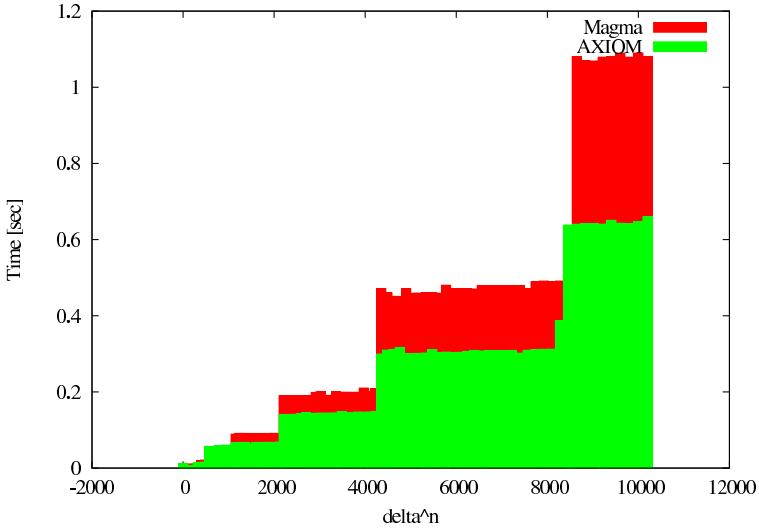

Figure 7.3: FFT-based bivariate polynomial multiplication modulo 18446744073692774401.

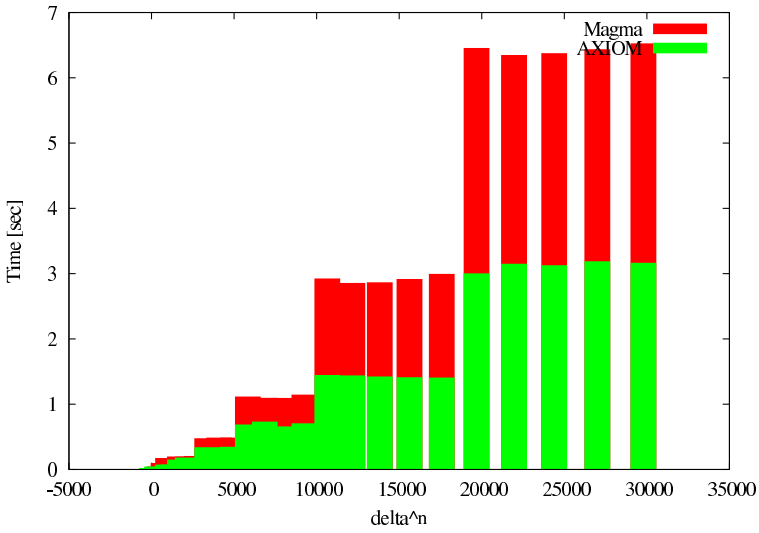

Figure 7.4: FFT-based trivariate polynomial multiplication modulo 18446744073692774401.

Chapter 8

Memory management

8.1 General idea

Many popular computer algebra systems have a garbage collector (GC) environment. Therefore, analyzing the efficiency of GC systems is another important issue in our research. We have chosen ALDOR as our experimental environment to study the impact of liveness information on the performance of a GC.

ALDOR's GC is one kind of tracing GC which employs the mark-and-sweep algorithm. Typically, a tracing GC collects objects that are no longer reachable on any path of references. This path of references starts from a set of roots, such as machine registers, run-time stacks, global variables and instruction pointers. It is well known that liveness information may aid in earlier reclamation of objects. It may help to reduce the set of root references, or remove dead references from the heap graph as soon as possible.

In this part of our work, we have concentrated on computing liveness information of all the local variables and parameters for each FOAM program (FOAM [39] is the intermediate language for ALDOR). Hence, this process is performed at compile-time in order to improve the memory management performance at run-time. The principle of our strategy is to modify the intermediate code as follows. When one local variable or parameter dies, we nullify it by explicitly assigning the `null` value to it. Thus, those ALDOR objects referenced by `null` could be identified at run-time

as unreachable and garbage collected soon.

According to [8], in order to compute the liveness information, an optimizing compiler can annotate each node n (or basic-block n) in the Control Flow Graph (CFG) (see [8] p. 439 for this notion) with a set called $LiveOut(n)$. This $LiveOut(n)$ set contains the names of all variables which are still alive when the control flow exits from the node n. A *live-out set* is defined by the following data-flow equations:

- $LiveOut(n_f) = \emptyset$ where n_f is the exit node of the CFG
- $LiveOut(n) = \bigcup_m (UEVar(m) \bigcup (LiveOut(m) \backslash Gen(m)))$ where m runs over the successors of n.

where

- $UEVar(m)$ contains the upward-exposed variables in m, that is the set those variables that are used in m before any redefinition happens,
- $Gen(m)$ set contains the variables that are defined in m.

The above equations encode the definition in an intuitive way. The set $LiveOut(n)$ is just a union of those variables that are still alive when exiting the basic block n (see [8] p. 440). By saying "alive" we mean a variable that will be referenced later in one or more paths.

Computing the *LiveOut* sets is performed by a three-step algorithm.

1. Building the CFG of the program. (This is already implemented in the Aldor compiler.)
2. Gathering the initial information for each basic-block; this involves computing the $UEVar$ and Gen sets.
3. Using an iterative fixed-point algorithm to propagate information around the CFG. In each iteration, this algorithm evaluates the data-flow equations. It halts when the information stops changing.

After obtaining the *LiveOut* sets for each basic-block, it is easy to deduce the so-called $CanKill(m)$ set at each node m. At node m, the set $CanKill(m)$ consists of the variables that are dead in the basic block m. After computing the the *CanKill* sets, we can insert the "assign-null" statements immediately at the corresponding locations.

8.2 Results

We have added the option $-Qcankill$ to the the ALDOR compiler in order to activate our transformation of the FOAM code described above. We give below an example of FOAM code transformation. The file `Fun01.as` is the original ALDOR source program and `Fun01.fm (part)` corresponds to a fragment of the FOAM code optimized by our transformation.

Fun01.as	Fun01.fm (part)
`#include "aldor"`	`(If(Cast Bool (Par 0 test))0)`
`import from Integer,`	`(Set(Loc 0 b)(Loc 1 a))`
`Boolean;`	`(Set(Loc 0 b)(Nil)) <- assign-null statment`
`f(test: Boolean): Integer==`	`(Label 1)`
`{`	`(Return (Loc 1 a))`
`a:=0; b:=1;`	`(Label 0)`
`if(test) then a:=b;`	`(Set(Loc 1 a)(Loc 0 b))`
`else b:=a;`	`(set(Loc 0 b)(Nil)) <- assign-null statement`
`a;`	`(Goto 1))))`
`}`	

Figure 8.1 illustrates the memory consumption without and with our package implementing the FOAM code transformation described above.

We can observe that the memory consumption is improved by our package.

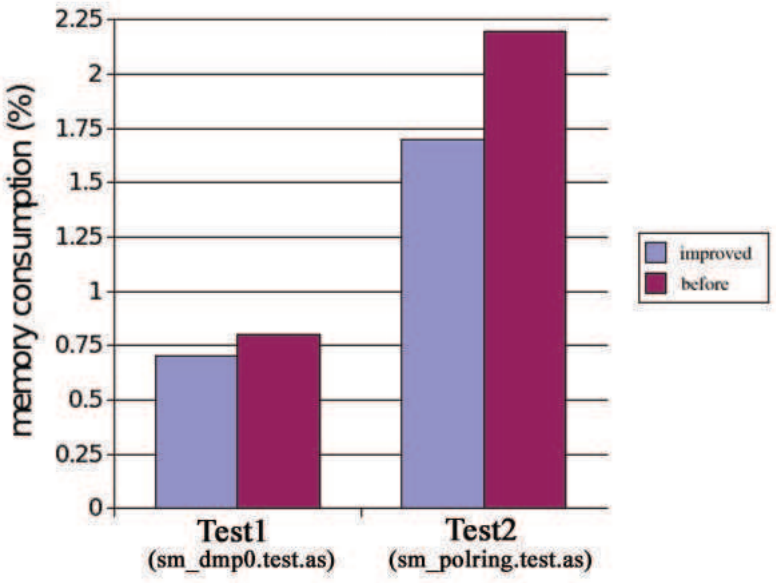

Figure 8.1: Testing the improvement of memory consumption

8.3 Future work of this chapter

To further improve the efficiency of our package, we:

- should use Reverse Post Order traversal (see [8] in P. 446) to propagate *LiveOut* information.
- should compute dominance information (see [8] in P. 457) to reduce the overall workload of data-flow analysis.

Chapter 9

Conclusions and future work

We have investigated and developed fast techniques for producing highly efficient polynomial arithmetic in AXIOM.

We have

1. combined high-level generic code and low-level machine dependent code in a transparent way for the high-level end-user,
2. developed adapted underlying data representations and algorithms,
3. implemented asymptotically fast polynomial arithmetic based on the Fast Fourier Transform,
4. investigated the impact of the garbage collection system on our applications.

For further improving the performance and competing with other efficient CAS such as MAGMA and NTL, we have also

1. studied the compilers' code optimization technique and its restrictions,
2. considered the hierarchy of the memory system,
3. used new features in IA-32 Intel Architecture.
4. employed parallel programming including instruction-level pipelining.

Our implementation results are satisfactory compared with other known CAS. Our implementation work can be ported to other CAS with a reasonable amount of effort.

In the near future, We will continue to adapt our polynomial packages to coefficient rings that are not necessarily fields but direct products of fields, [29] [30] and interface our Assembly code and C libraries with other CAS or languages, such MAPLE and ALDOR.

Bibliography

[1] *The* AXIOM *developers web site*. `http://page.axiom-developer.org`.

[2] aldor.org. *The* ALDOR *distribution website*. `http://www.aldor.org`.

[3] Randal E. Bryant and David O'Hallaron. *Computer Systems*. Pearson Education, Inc., 2003.

[4] David G. Cantor and Erich Kaltofen. On fast multiplication of polynomials over arbitrary algebras. *Acta Informatica*, 28:693–701, 1991.

[5] The CoCoA team. *The CoCoA Computer Algebra System*. `http://cocoa.dima.unige.it/`.

[6] Henri Cohen and his co workers. *PARI/GP*. `http://pari.math.u-bordeaux.fr/`.

[7] 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/`.

[8] Keith D. Cooper and Linda Torczon. *Engineering A Compiler*. Morgan Kaufmann, 2004.

[9] Martin Cracauer. CMUCL*: a high-performance, free Common Lisp implementation*. `http://www.cons.org/cmucl/`.

[10] Xavier Dahan, Marc Moreno Maza, Éric Schost, Wenyuan Wu, and Yuzhen Xie. Lifting techniques for triangular decompositions. In *Proc. ISSAC'05*. ACM Press, 2005.

[11] David A. Cox, John B. Little and Donald O'Shea. *Ideals, varieties, and algorithms: an introduction to computational algebraic geometry and commutative algebra*. Springer, 1997.

[12] David S. Dummit and Richard M. Foote. *Abstract algebra*. Simon and Schuster, 1993.

[13] Ioannis Z. Emiris and Victor Y. Pan. Fast fourier transform and its applications. In Mikhail J. Atallah, editor, *Handbook of Algorithms and Theory of Computations*. CRC Press Inc, 1999.

[14] Richard J. Fateman. Vector-based polynomial recursive representation arithmetic. 1990. `http://www.norvig.com/ltd/test/poly.dylan`.

[15] Richard J. Fateman. Can you save time in multiplying polynomials by encoding them as integers? 2005. `http://http.cs.berkeley.edu/~fateman/papers/polysbyGMP.pdf`.

[16] Free Software Foundation. *GNU Multiple Precision Arithmetic Library*. `http://swox.com/gmp/`.

[17] Joachim von zur Gathen and Jurgen Gerhard. *Modern Computer Algebra*. Cambridge University Press, 1999.

[18] K.O. Geddes, S.R. Czapor, and G. Labahn. *Algorithms for Computer Algebra*. Kluwer Academic Publishers, 1992.

[19] Linley Gwennap. Intel's p6 uses decoupled superscalar design. In Microprocessor Report, Vol. 9, Issue 2, February 1995.

[20] Mark van Hoeij and Michael B. Monagan. A modular gcd algorithm over number fields presented with multiple extensions. In Teo Mora, editor, *Proc. ISSAC'02*, pages 109–116. ACM Press, July 2002.

[21] Joris van der Hoeven. Truncated fourier transform. In *Proc. ISSAC'04*. ACM Press, 2004.

[22] Richard D. Jenks and Robert S. Sutor. *AXIOM the Scientific Computation System*. Springer-Verlag, 1992.

[23] Jeremy R. Johnson, Werner Krandick, and Anatole D. Ruslanov. Architecture-aware classical Taylor shift by 1. In *Proc. ISSAC'05*. ACM Press, 2005.

[24] A. Karatsuba and Yu. Ofman. Multiplication of multidigit numbers on automata. *Soviet Physics Doklady*, (7):595–596, 1963.

[25] Grégoire Lecerf and Éric Schost. Fast multivariate power series multiplication in characteristic zero. *SADIO Electronic Journal on Informatics and Operations Research*, 5(1), September 2003.

[26] Xin Li, Marc Moreno Maza, and Éric Schost. Fast multivariate polynomial multiplication. 2005. preprint.

[27] Marc Moreno Maza. On triangular decompositions of algebraic varieties. Technical Report TR 4/99, NAG Ltd, Oxford, UK, 1999. Presented at the MEGA-2000 Conference, Bath, England.

[28] Marc Moreno Maza. CS 874 advanced computer algebra: asymptotically fast methods for exact computations. 2002. `http://www.csd.uwo.ca/~moreno//MainPages/CS874-2003.html/index.html`.

[29] Marc Moreno Maza and Cosmin Oancea. On polynomial gcds over products of fields presented by towers of simple extensions. Technical report, University of Western Ontario, January 2004.

[30] Marc Moreno Maza and Renaud Rioboo. Polynomial gcd computations over towers of algebraic extensions. In *Proc. AAECC-11*. Springer, 1995.

[31] Victor Y. Pan. Simple multivariate polynomial multiplication. *J. Symb. Comp.*, 18(3):183–186, 1994.

[32] William F. Schelter. AKCL*: The* Austin Kyoto Common LISP *compiler*. `http://hpux.connect.org.uk/hppd/hpux/Languages/akcl-1.619/`.

[33] É. Schost. Complexity results for triangular sets. *J. Symb. Comp.*, 36(3-4):555–594, 2003.

[34] Victor Shoup. NTL*: A Library for doing Number Theory*. `http://www.shoup.net/ntl/`.

[35] Victor Shoup. A new polynomial factorization algorithm and its implementation. *J. Symb. Comp.*, 20(4):363–397, 1995.

[36] Richard M. Stallman and the GCC developement team. *the GNU Compiler Collection, includes front ends for C, C++, Objective-C, Fortran, Java, and Ada*. `http://gcc.gnu.org/`.

[37] Henry S. Warren, Jr. *Hacker's Delight*. Pearson Education, Inc., 2003.

[38] Stephen M. Watt. A# language reference v0.35. Technical Report RC 19530, IBM Research, 1994.

[39] Stephen M. Watt. FOAM: A first order abstract machine. Technical report, IBM Research, 1994.

[40] Taiichi Yuasa, Masami Hagiya, and William F. Schelter. *GNU Common Lisp*. `http://www.gnu.org/software/gcl`.

[41] Richard Zippel. *Effective Polynomial Computation*. Kluwer Academic Press, 1993.