Towards an Implementation of a Computer Algebra System in a Functional Language

Oleg Lobachev and Rita Loogen

Philipps–Universität Marburg, Fachbereich Mathematik und Informatik
Hans–Mehrwein–Straße, D–35032 Marburg, Germany
{lobachev,loogen}@informatik.uni-marburg.de

Abstract. This paper discusses the pros and cons of using a functional language for implementing a computer algebra system. The contributions of the paper are twofold. Firstly, we discuss some language–centered design aspects of a computer algebra system — the "language unity" concept. Secondly, we provide an implementation of a fast polynomial multiplication algorithm, which is one of the core elements of a computer algebra system. The goal of the paper is to test the feasibility of an implementation of (some elements of) a computer algebra system in a modern functional language.
Keywords: computer algebra, software technology, language and system design.

1 Introduction

With the flow of the history of computing, exact methods gained more and more importance. It was clear since almost the beginning, that imprecise, *numerical* operations may and will fail. The Wilkinson Monster $\prod_{j=1}^{20}(x-j)$ is a nice – and old! [45,46] – example for the thesis "the way we compute it matters". One of the crucial points of computer algebra systems (CAS) is the implementation of fast algorithms. One of the core algorithms is fast multiplication, be it of numbers or of polynomials. Current approaches include methods by Karatsuba, Toom and Cook [20, 44, 24] and Schönhage and Strassen [38, 37]. An implementation of the latter in the functional language `Haskell` [33] is presented in this paper to test the suitability of functional languages for implementing computer algebra algorithms. Our vision is an open-source flexible computer algebra system, that can easily be maintained, extended and optimised by the computer algebra community. The mainstream computer algebra systems like `Maple` [34] or `Mathematica` [47] provide highly optimised routines with interesting but hidden implementation details. However, the closed–source nature of such systems does not enable us to analyse their internals. On the contrary, the following modern CAS are examples for systems with freely available source code: `CoCoA` [8], `DoCon` [27], `GAP` [11], and `GiNaC` [16, 13]. Our approach follows the philosophy of the `GiNaC` library, which extends a given language (`C++`) by a set of algebraic capabilities, instead of inventing a separate interface language for that purpose.

We plan to implement a computer algebra system in a modern functional language like `Haskell`. Several features of such languages, like lazy evaluation, improve numerical computations [5,6]. Lazy evaluation is also helpful for designing algorithms in scientific computing [22]. Other features could as well be useful for a CAS [28]. Incidentally, both functional programming languages [25,42,29,2,35] and computer algebra systems [15, 14, 18, 39] are present in the field of parallel and distributed computing.

Plan Of The Paper

The second section discusses the benefits of functional languages for implementing computer algebra algorithms. Section 3 pushes the *language unity concept* for CAS, i.e. choosing the same language for implementing and using a CAS. Section 4 presents a few case studies. We

a) compare different `Haskell` implementations of polynomial multiplication,
b) compare `Haskell` and imperative implementations for computing factorials,
c) consider the FFT–based implementation of polynomial multiplication by Schönhage and Strassen.

Section 5 concludes the paper. Code samples are presented in Figures 3 and 4 in Section 4.

2 Advantages of Functional Languages

We consider `Haskell` [33] as a base of our thoughts. Some of the key features of most functional programming languages, all of them found in `Haskell`, are:

- *Lazy evaluation* means that no expression is evaluated if it is not required. This can be combined with *memorisation*, when no expression is evaluated more than once. We should think of lazy evaluation as of a double–edged sword. Indeed it reduces the amount of required computations and the end user of the CAS has the freedom of writing his/her own programs in a way more corresponding to standard mathematical nomenclature. However, worse performance will be observed, if lazy evaluation fails to outweigh its overhead by skipping evaluations. A detailed comparison is beyond the scope of this paper. However nice applications of lazy evaluation in the context of scientific computing can be found in papers by Jerzy Karczmarczuk [21, 22, 23]
- Functional languages provide *infinite data structures*, notably: lists. Such lists can be easily implemented with lazy evaluation. Infinite data structures enable "more mathematical" definitions of e. g. sequences and series. On the one hand, this means "more conforming to the current mathematical nomenclature" as in e.g. `factorial n = product [1..n]` and, on the

other hand, "nice in describing typical mathematical concepts" including infinite sequences. A classical example for this is `fibs = 0 : 1 : zipWith (+) fibs (tail fibs)`[1].

- *Referential transparency* enables a "more mathematical" semantics: for function `f`, `f(5)` has the same value, whenever it is evaluated, pretty much as $f(5)$ in a mathematical notation. Consider an example in `C`.

  ```
  int i = 5;
  i = ++i + i++;
  ```

  This example is rather unnatural, but the result value of `i` depends on the implementation – try it in any imperative language of your choice. In a pure functional language, such dubious definitions are not possible.
- In the context if a CAS *strong typing* gives some benefits. For example, it is possible to produce an error at compile time for a product of matrices of incompatible dimensions. On the other hand, *type inference* is possible. However there are some problems with `Haskell` type system in a computer algebra context. For instance, if you define a factor ring over a commutative ring, it may or may be not a field: it depends on the properties of the ideal. If rings, domains, etc. are defined as types, the `Haskell` type system would not be able to determine at compile time, whether this *instance* of type "factor ring" is a field or not. Papers by S. Mechveliani, for instance [26], discuss this problem and suggest an appropriate solution.
- `Haskell`'s *hierarchical module system*, being a rather software engineering issue, provides the possibility to structure large programs efficiently.
- Another benefit of modern functional languages is the possibility to *prove* the correctness of implementations.

3 The Two Languages of a CAS

Computer algebra systems possess two different languages, we shall call them in this paper as follows. The *internal* language of a CAS is the language the system is written in, the implementation language. Since the end user of the CAS wants to perform some kind of *programming*, there is also a second language. The *external* language of a CAS is the language for user interaction, the interface language. The idea of "language unity" is to utilise the same language for both purposes, i.e. as internal and as external language.

It is desirable to write as much as possible of the CAS itself in its external language. This gives the user the opportunity to inspect and (if needed) to modify some external functions of the CAS. However, for several reasons, this is impossible in most CAS. Firstly, the external language of most CAS is "weaker" than their internal one in the sense that some technical things may be hard or even impossible. On the other hand, the external language is better suited for the typical computer algebra operations: we may expect, e. g. polynomials and matrices as native objects or an interesting handling of lists, non–existent

[1] See `http://haskell.org/haskellwiki/The_Fibonacci_sequence` for a sublinear time implementation of the same sequence.

in the internal language of the CAS if this language is imperative. Especially advanced features like type safety and generic programming are desired in the external language. A recent development is to utilise a general purpose dynamic language like `Ruby` [10], `Groovy` [3] or `Python` [43] for interconnecting different programs, building a composite computer algebra system [40].

Secondly, unfortunately, the external language of most CAS is not as fast as the internal one. The cause may be the interpreted origin of these languages or their very high level nature. This is often avoided by compiling the input files to some kind of byte code. Other speedup approaches compromise the extensibility. The implementation of the `S` programming language for statistical computations, `GNU R`, utilises a `Scheme` dialect as its external language. The whole R system could be implemented in `Scheme`. But because of performance lack in core operations, these are replaced with function calls from the bundled `C` library. These functions can still be overloaded and replaced by the user's own version, but one cannot simply look into the routines, which are sped up this way. There is also a third option: to use a functional language and to perform optimisations in the language compiler typical for a functional language. This way our external language could be feature–rich and reasonably fast, but it will have the price of writing a, say, `LISP` interpreter in an imperative language.

An interesting approach in this field was taken by Christian Bauer, Alexander Frink, Richard Kreckel et al., the developers of `GiNaC` [4, 13]. This computer algebra system was written in `C++` and it maintains `C++` as its main interface. It is made in a very simple way: `GiNaC` is rather a computer algebra *library*, than a complete system. So the primary use of `GiNaC` is to give one a possibility of writing his/her own `C++` programs, while using arbitrary precision numbers, polynomials, matrices, expression evaluation and other nice and fast computer algebra functions, offered by the `GiNaC` library. As the authors of `GiNaC` state:

> Its design is revolutionary in a sense that contrary to other CAS it does not try to provide extensive algebraic capabilities and a simple programming language, but instead accepts a given language (`C++`) and extends it by a set of algebraic capabilities.

This approach is very interesting and powerful, but the interactive front end program of `GiNaC`, the `ginsh`, is less powerful due to a rather weak language. It was, however, never intended to be a complete `GiNaC` interface. The possibility to use all the `GiNaC` features at an interactive prompt requires a `C++` *interpreter*. While interpreting `C++` is not very nice (although possible: see e.g. [9])[2], it is much easier with `Haskell`: aside from the glorious Glasgow Haskell Compiler [12], we have Hugs, the `Haskell` interpreter. Also GHC itself offers an interactive version, GHCi. The latter is capable of loading pre-compiled object files into the interpreted environment. With this achievement one has the possibility to write a computer algebra system, whose *external* interface language equals its *internal* implementation language, and this language is a functional one.

[2] There is also a third party `GiNaC` interface language project, `http://swiginac.berlios.de/`.

The idea of `GiNaC` was not born in vain: most *long* CAS–supported computations are run in "batch mode", with no user interaction. It seems plausible not to wait in front of a command prompt for the result for hours, days or even months.[3] On the other hand, most of CAS–based *development* is done in an interactive environment, in a "shell". If one could use the same language both for developing and for lengthy computations, this would be a major success in saving developers' work time [32] and gaining stability of computations.

Now why not just make both: a compiler *and* an interpreter of CAS' external language? The problem is, that despite many efforts, the external languages of computer algebra systems are slow. On the other hand, we already have a fast language in our CAS–developing project. This is the language, the CAS itself is written in, the *internal* language. One may oppose, however, the whole game with computer algebra system's *external* language was started, because the internal language was not high–level enough for vectors, matrices, polynomials and all the other expressions, which are eagerly wanted in a full–fledged CAS. Now we come back to the beginning of this paper. Functional languages *are* complicated and high–level enough to have all the aforementioned objects and properties [33,41,27,7,17]. Functional languages have very compact code size and rapid development times [32]. Most functional languages have very interesting data structures and language design features, which benefit both featuring them as an internal or as an external language, see [31] for details. And some modern functional languages already have an efficient compiler and an interpreter implemented, which leads us to the future goal of internal and external language fusion. `Haskell` is an example of a such language.

Concluding: an implementation of a CAS in a functional language utilising the above "language unity" concept will greatly reduce code size and improve readability, at the same time it shall not reduce the performance significantly. In order to test the feasibility of these assumptions we consider several case studies.

4 First Case Studies

Now as we have seen some *theoretical* reasons for a CAS to be implemented in `Haskell`, let's take a look at some examples. At first we shall examine the univariate polynomials. One can hardly imagine a computer algebra system without them, polynomials are used in thousands of higher–level algorithms and the operations with the polynomials should be fast. Unfortunately as of today neither of available `Haskell` software packages implementing univariate polynomials uses sub–quadratic algorithms like Karatsuba[4], Toom–Cook or Schönhage–Strassen algorithms. As one of the examples we demonstrate an implementation

[3] In this case one might think of porting his/her CAS–based program to some low–level language and let, say, `FORTRAN` run the number–crunching mills. However this is a highly interactive and bug–ridden process. And the `FORTRAN` program is to be tested for errors *again*, before the real computations may begin: the thoroughly tested CAS–routines are not enough!

[4] Although an implementation of this algorithm in `Haskell` was presented in [19, 36].

of Schönhage–Strassen algorithm in `Haskell`. But first we look at the schoolbook case.

All the tests were run on the same machine[5] with the same compiler – GHC 6.8.2. For the same n, each test was run ten times and the mean value of measured execution time has been determined. We utilise standard `Haskell` lists for representing the polynomials. The complete system would use some kind of generalisation layer, probably based on type classes, to abstract the implementation from the given representation. It would be sufficient to redefine the few standard functions on lists to obtain the implementation of the same algorithm for yet another data structure. No modification of the presented code would then be required.

4.1 Naive Polynomial Multiplication

We have tested four different $\mathcal{O}(n^2)$ implementations:

1. our own naive implementation with lists of integers
2. our naive implementation, modified à la Numeric Prelude,
3. the implementation from *Haskell for Math* [1],
4. the implementation from *Numeric Prelude* [41].

We multiply two dense univariate $(n-1)$–grade polynomials with random coefficients. The coefficients are random signed 32–bit integers: what we test here are the polynomial multiplication implementations, not the hardware multiplication of small integers, nor even different libraries for arbitrary precision integers. Nor do we test the quadratic algorithms – they all represent pretty the same "school" multiplication – or compiler options, but the impact of the particular implementation decisions on the performance. The naive implementation uses a "dumb" list of `Int`s, the other implementations build a chain of types similar to the algebraic objects. One can e.g. define addition and subtraction for elements of the additive group, multiplication for elements of this group embedded into a ring, and finding an inverse for invertible elements of this ring embedded into a field. An overview of test results is provided in Figure 1. Time is measured in seconds. The Numeric Prelude implementation is much better than the other implementations which show similar runtimes. Note that the simplest implementation is *not* the fastest one and that the type hierarchy enables optimisations. Nevertheless, we conclude the strong need for sub–quadratic implementations.

4.2 Computing Factorial

We would like to discuss briefly another example. We take a well–known and very quickly growing function on integers: the factorial. We have tested the famous `Haskell` one–liner `factorial n = product [1..n]`, and two `C++` implementations. Both `C++` versions are based on the CLN [16] – the arbitrary precision library used in `GiNaC`. One implementation uses the built–in factorial function

[5] AMD Athlon 64 X2 4000+ CPU with 1 Gb RAM, running Gentoo Linux.

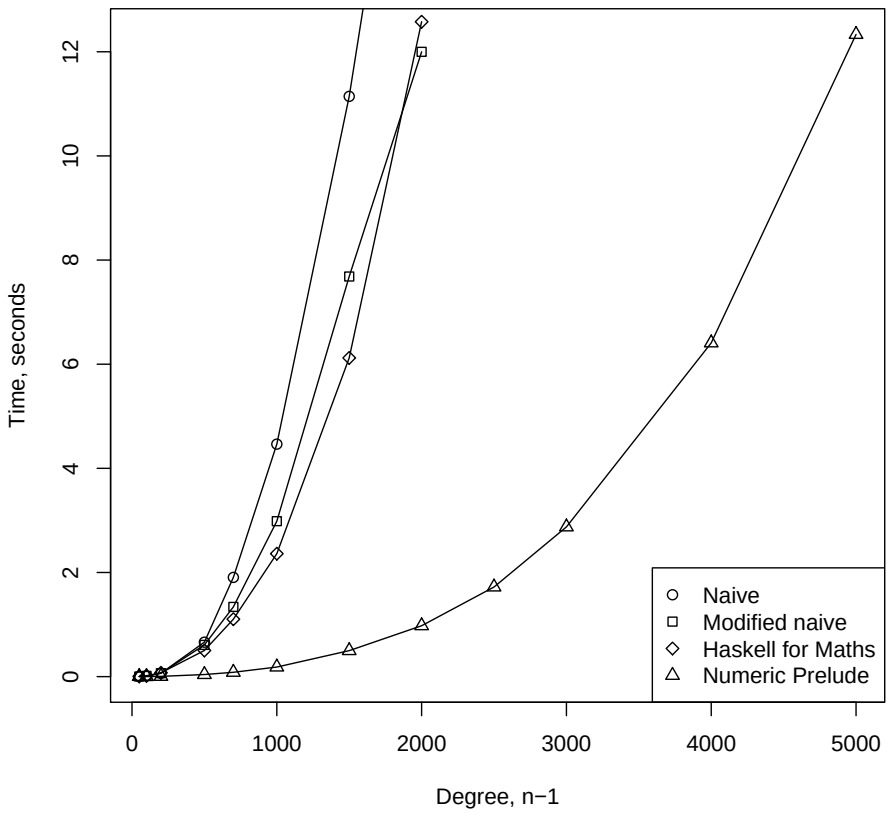

Fig. 1. Multiplication of univariate polynomials of degree $n-1$.
Runtime comparison of naive implementations.

from the CLN. It makes use of table look–ups and computes some parts of the factorial value in divide and conquer fashion. The other `C++` implementation is not optimised, but it still uses CLN built–in multiplication and large integers. We find this implementation comparable with the naive `Haskell` implementation. Arbitrary long integers are provided in `Haskell` out of the box. We are not willing to discuss the details of arbitrary precision arithmetic implementation in `Haskell` compiler runtime, our focus is to demonstrate how competitive the functional approach is. The graphical representation of the obtained results is shown in Figure 2. The timings of the `Haskell` version lie in between both `C++` versions.

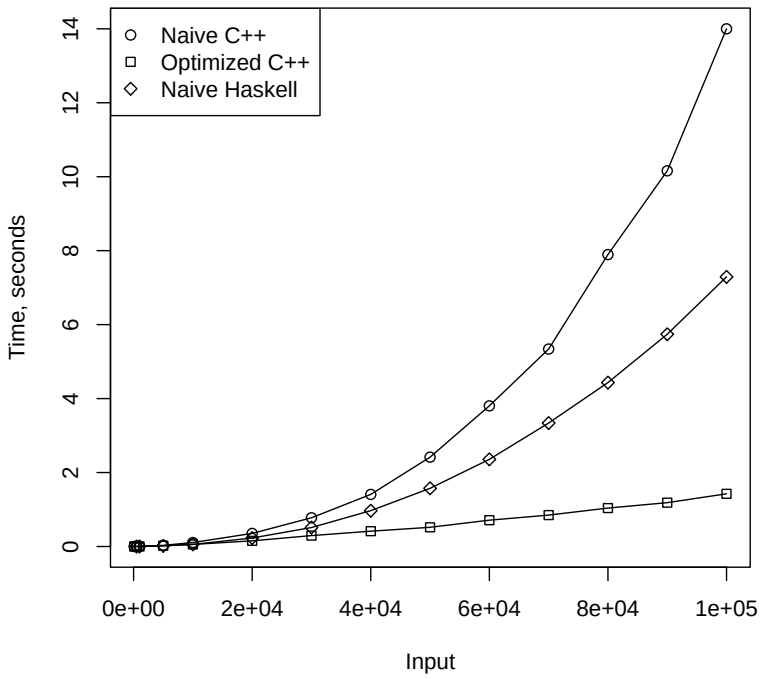

Fig. 2. Computing the factorial.

This small example shows that `Haskell` implementations, even in their simplest and primitive form are competitive with implementations in some industry–used programming language which are more sophisticated in programming effort. The optimised version outperforms both naive versions, thus motivating us to create implementations of fast algorithms in `Haskell`.

4.3 Fast Polynomial Multiplication

The essence of Schönhage and Strassen's method for fast polynomial multiplication[6] is the way a *convolution* is performed. A convolution in $\mathbb{C}[x]$ corresponds to multiplication, as in "each with every". A convolution in Fourier–transformed

[6] ... over the domains supporting the fast Fourier transform, just like complex numbers $\mathbb{C}$. If the domain does not support FFT, the fast multiplication is still possible, through an implicit algebraic extension of the original domain. For details please refer to the original paper [38] or a standard book on this topic [44].

```
fft     :: [Complex Double] -> [Complex Double]
fft f   =  mix [fft (l @+ r), fft ((l @- r)@* w)]
           where (l, r)   = splitAt (length f `div` 2) f
                 mix      = concat . transpose
                 (@+) f g = zipWith (+) f g  -- @-, @* analog
                 -- w is list of powers of an n-th primitive root of unity.
```

Fig. 3. Implementation of Cooley–Tukey algorithm in `Haskell`.

```
(%*%)     :: (Num a) => [a] -> [a] -> [a]
(%*%) f g =  unlift $ ifft ((fft $ lift f) @* (fft $ lift g))
             -- where lift   :: (Num a) => [a] -> [Complex Double]
             --       unlift :: [Complex Double] -> [Int]
             -- ifft is the inverse fft, basicly the same fft with
             -- different twiddle factors.
             -- And (@*) is still element-wise multiplication
```

Fig. 4. FFT–based multiplication modulo $x^n - 1$ in `Haskell`.

space is just a component–wise multiplication. So if we want to compute a product of two polynomials, we compute their Fourier transformed (e. g. with the routine in Figure 3), then multiply the transformed functions component wise and then, with the inverse Fourier transformation, transform the product back to a polynomial (Figure 4). The presented version performs twice as well as the full version at the price of not computing the complete product. However, the current implementation for computing the full product can be easily obtained from this code. The functions `zipWith`, `splitAt`, `length`, `concat` and `transpose` are provided by `Haskell` standard libraries. `zipWith` "zips" two lists with a supplied binary function, e.g. `zipWith (+) [1,2,3] [4,5,6]` results in `[5,7,9]`. `splitAt` splits a list into two parts at the provided offset. `length` returns the length of a list. `concat` concatenates a list of lists to a list. `transpose`, as the name says, transposes a list of lists. The functions `lift`, `unlift`, `ifft` and `(@*)` are part of our implementation. The inverse Fourier transformation is nothing spectacular and is pretty much the forward Fourier transformation with different values. As the fast Fourier transform (FFT) for a polynomial in $\mathbb{C}[x]$ of degree $n-1$ can be performed in $\mathcal{O}(n \log n)$ time and the component–wise multiplication in $\mathcal{O}(n)$, we can multiply two polynomials of degree $n-1$ in $\mathbb{C}[x]$ in $\mathcal{O}(n \log n)$ time [44]. Due to limitations of the naive implementation we receive the remainder of the product after the division through $x^n - 1$. But it is still possible to compute the whole product without changing the asymptotic complexity, for example, applying one step of the Karatsuba algorithm first, or just padding both arguments to the length of the product.

The technical representation of a polynomial in our case is a list of coefficients. The Cooley–Tukey decimation in frequency algorithm is utilised, using a divide–and–conquer approach for computing the Fourier transform. This is the

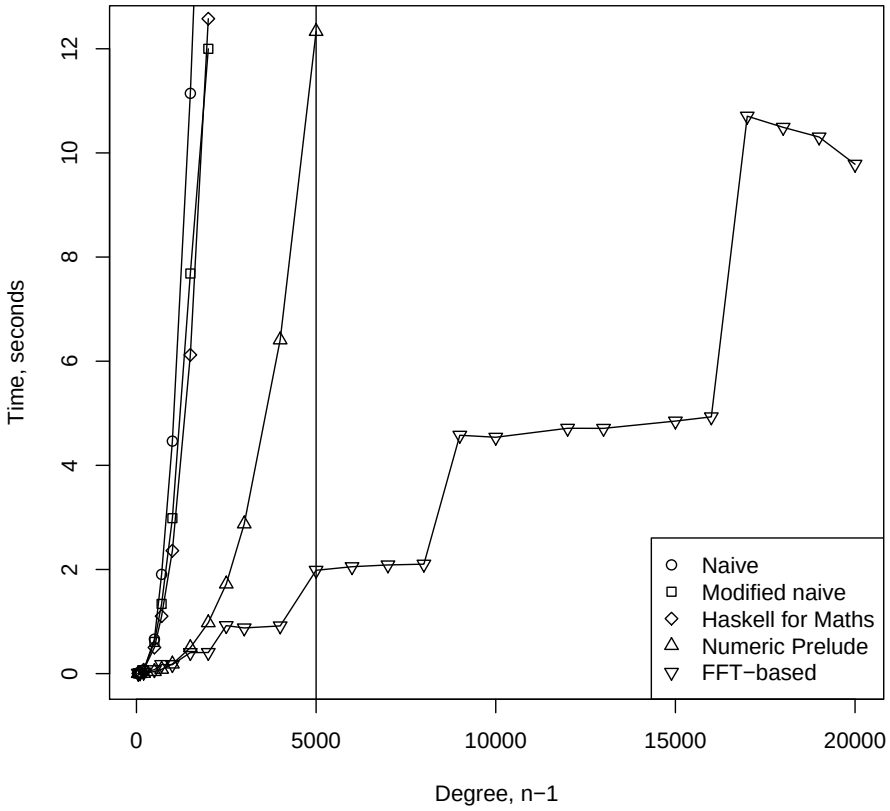

Fig. 5. Multiplication of dense univariate polynomials of degree $n-1$ revised. Naive Implementations vs. FFT–based. The left side of the plot corresponds to the Figure 1.

simplest FFT algorithm, there exist some more sophisticated variants [44, 30]. Figure 5 presents the results, we used the same kind of input as in Figure 1. A sub–quadratic method for polynomial multiplication is definitely superior. The bottom line is the FFT–based multiplication algorithm, we compute the whole product. It is clearly visible, that the current FFT algorithm is relying on the fact, that the length of its input is a power of two. The rapidly ascending lines correspond to the values shown in Figure 1. Unfortunately, we have no explanation for the decreasing values of the FFT-based algorithm for $n \in [16000..20000]$.

Now we have seen a fast polynomial multiplication in Haskell. By using advanced algorithms we significantly increase the performance, the implemented functions can be used by any other Haskell program, as we have not tweaked the compiler. The size of the code base is modest for the task it accomplishes. This case study shows that it is possible to extend Haskell with further implementations of fast computer algebra algorithms, obtaining in the end a computer algebra library. The main interface to this system is the language itself, direct interaction with the library is possible with an interpreter.

5 Related Work

Writing a computer algebra system in a functional programming language is not a really new idea. The first generation CAS named Macsyma was written in LISP 1.5 dialect called MACLISP, and LISP is considered to be the first functional language ever. Axiom CAS has some interesting aspects. It features an embedded (although detachable) functional programming language [7]. In addition, it uses a hierarchical structure of mathematical objects (like: monoid – group – ring – integrity domain – field) to specify and perform operations on them.

The DoCon computer algebra library [27] is at the first glance very similar to our intention. It utilises Haskell as implementation language. Being a library, it also has Haskell as an interface language. However, DoCon pursues a different goal. DoCon is an algebra *framework*, implementing different mathematical objects and their relations, thereby heavily dependent on Haskell's type system. For instance, it is easy to define a residue domain modulo some polynomial ideal in DoCon. However, we focus on the computer algebra *algorithms*. We would like to have e.g. a fast polynomial multiplication, while representing the polynomials as simply as possible. Moreover, we are interested in *parallelising* our algorithm implementations. Because of high communication costs, we need to keep the underlying data structures as "dumb" as possible. It will be interesting to utilise the DoCon approach in our own work and to share our results with the current DoCon implementation.

6 Conclusions and Future Work

We propose to unify the internal implementation and the external interface language of computer algebra systems and to use a functional language to achieve this integration. The usage of a functional language in a computer algebra field drastically reduces the size of the source code. Secondly, it does not affect the performance. Hence, is not required to mix two different languages in an implementation of a CAS. We have shown that functional programs are competitive with mainstream imperative programs and significantly easier to develop.

Concerning the performed case studies, a possible direction of the future work would be the optimisation of the fast Fourier transform. Some practical tests in the parallel context indicate an optimisation potential in switching to decimation in time. From the theoretical viewpoint, it will be interesting to reconstruct the

Fourier transformed values in special cases, the so–called pruned FFT algorithm. It would also be of interest to try other FFT algorithms, for example, the r–radix implementations.

Concerning the future goals of this work: Modern functional languages and computer algebra are two rapidly developing research areas, an intersection of these two areas is highly interesting. A third component to mix into this "cocktail" of computational algebra and functional programming topics is parallelism. Computer algebra applications tend to be quite resource hungry and functional languages have great potential in parallelism, which is being currently quite extensively investigated. With respect to our gradually evolving practical implementation, modern algorithms of computer algebra should be implemented in relevant `Haskell` software packages, as a naive implementation typically leads to asymptotically bad complexity. One should carefully design such implementations, as design choices play a significant role for the execution times in the same complexity class. Such choices gain even more on importance in the parallel setting. The aforementioned algorithms should provide

- fast polynomial multiplication – tackled in this paper,
- fast integer multiplication – our current approach is to use fast polynomial multiplication,
- efficient Euclid's algorithm for polynomials,
- efficient vector and matrix computations,
- framework for symbolic computation and object manipulation.

Such foundation will be a solid base for more complex research areas, including

- algorithms of numerical number theory,
- implementation of public key cryptography,
- algorithms of computational algebraic geometry, based on Gröbner bases,
- symbolic integration and summation,
- parallel computations.

As for FFT–based multiplication, we provide our `Haskell` implementation of polynomial multiplication, a multiplication routine for arbitrary long integers based on top of it and an interface script to SCSCP [39] on request.

Acknowledgement

We would like to thank to anonymous referees for their helpful and detailed comments.

References

1. David Amos. Haskell for Math program. `http://www.polyomino.f2s.com/david/haskell/codeindex.html`.
2. Joe Armstrong. *Programming Erlang.* The Pragmatic Programmers, LLC, 2007.

3. Kenneth Barclay and John Savage. *Groovy Programming: An Introduction for Java Developers*. Morgan Kaufmann Publishers Inc. San Francisco, CA, USA, 2006.
4. Christian Bauer, Alexander Frink, and Richard Kreckel. Introduction to the GiNaC Framework for Symbolic Computation within the C++ Programming Language. *J. of Symbolic Computation*, 33:1–12, 2002.
5. M. O. Benouamer, D. Michelucci, and B. Peroche. Error-free boundary evaluation based on a lazy rational arithmetic: a detailed implementation. *Computer Aided Design*, 26(6):403–416, 1994.
6. Richard S. Bird, Geraint Jones, and Oege De Moor. More haste, less speed: lazy versus eager evaluation. *J. of Functional Programming*, 7(5):541–547, 1997.
7. Manuel Bronstein, James Davenport, Albrecht Fortenbacher, et al. Axiom – the 30 year horizon, 2003. `http://portal.axiom-developer.org/public/book2.pdf`.
8. A. Capani and G. Niesi. *CoCoA 3.0 User's Manual*. Dipartimento di Matematica, Università di Genova, Via Dodecaneso, 35, I-16146 Genova (Italy), 1995.
9. Cint, the C/C++ interpreter, version 5.16.19. `http://root.cern.ch/root/Cint.html`.
10. David Flanagan and Yukihiro Matsumoto. *The Ruby Programming Language*. O'Reilly, 2008.
11. The GAP Group. *GAP – Groups, Algorithms, and Programming, Version 4.4.10*, 2008.
12. The Glorious Glasgow Haskell Compilation System User's Guide. `http://www.haskell.org/ghc/docs/latest/users_guide.pdf`, February 2008.
13. GiNaC program. `http://www.ginac.de`.
14. HPC-Grid for Maple program. `http://www.maplesoft.com/products/toolboxes/HPCgrid/index.aspx`.
15. gridmathematica2 program. `http://www.wolfram.com/products/gridmathematica/`.
16. Bruno Haible and Richard Kreckel. *CLN, a class library for numbers manual*, 2005. `http://www.ginac.de/CLN/cln.ps`.
17. Cordelia Hall, Kevin Hammond, Simon Peyton Jones, and Philip Wadler. *European Symposium On Programming*, volume LNCS 788, chapter Type classes in Haskell, pages 241–256. Springer, April 1994.
18. K. Hammond, A. Al Zain, G. Cooperman, D. Petcu, and P. Trinder. Symgrid: a framework for symbolic computation on the grid. LNCS 4703. EuroPar'07 – European Conf. on Parallel Processing, Rennes, France, Spinger-Verlag, August 2007.
19. Christoph A. Herrmann and Chrisitan Lengauer. HDC: A Higher–Order Language for Divide–and-Conquer. *Parallel Processing Letters*, 10(22):239–250, 2000.
20. A. Karatsuba and Yu. Ofman. Multiplication of many-digital numbers by automatic computers. *Doklady Akad. Nauk SSSR*, 145:293–294, 1962. Translation in Physics–Doklady 7, 595–596, 1963.
21. Jerzy Karczmarczuk. The most unreliable technique in the world to compute pi, 1998.
22. Jerzy Karczmarczuk. Scientific computation and functional programming. *Computing in Science & Engineering*, 1(3):64–72, May/June 1999.
23. Jerzy Karczmarczuk. Functional differentiation of computer programs. *Higher–Order and Symbolic Computation*, 14(1):35–57, 2001.
24. Donald E. Knuth. *The Art of Computer Programming*, volume 2. Addison–Wesley, third edition, 1998.
25. Rita Loogen, Yolanda Ortega-Mallén, and Ricardo Peña-Marí. Parallel Functional Programming in Eden. *Journal of Functional Programming*, 15(3):431–475, 2005.

26. Serge D. Mechveliani. Haskell and computer algebra. Manuscript, 2000. Pereslavl-Zalessky, Russia.
27. Serge D. Mechveliani. *DoCon. The Algebraic Domain Constructor Manual.* Program Systems Institute, Pereslavl–Zalessky, Russia, 2007. Version 2.11.
28. Gérard Milmeister. Functional kernels with modules. Master's thesis, ETH Zürich, 1995.
29. R. S. Nikhil, L. A. Arvind, J. Hicks, S. Aditya, L. Augustsson, J. Maessen, and Y. Zhou. *pH Language Reference Manual, Version 1.0.* Massachusetts Institute of Technology, 1995. Computation Structures Group Memo No. 396.
30. H. J. Nussbaumer. *Fast Fourier Transform and Convolution Algorithms.* Springer, Berlin, 1981.
31. Chris Okasaki. *Purely Functional Data Structures.* Cambridge University Press, 1998.
32. M. P. Jones P. Hudak. Haskell vs. Ada vs. C++ vs. awk vs.... An experiment in software prototyping productivity. Yale University, Department of Computer Science, July 1994.
33. Simon Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report.* Cambridge University Press, December 2003.
34. Darren Redfern. *The Maple Handbook: Maple V Release 4.* Springer, December 1995.
35. Peter Van Roy, editor. *Multiparadigm Programming in Mozart/Oz.* Second International Conference, MOZ, 2004.
36. Christian Schaller. Elimination von Funktionen höherer Ordnung in Haskell–Programmen. Master's thesis, Universität Passau, September 1998.
37. Arnold Schönhage. Asymptotically fast algorithms for the numerical multiplication and division of polynomials with complex coefficients. volume 144 of *Lect. Notes Comp. Sci.*, pages 3–15. EUROCAM '82: European Computer Algebra Conference (Marseille, France), April 1982.
38. Arnold Schönhage and Volker Strassen. Schnelle Multiplikation großer Zahlen. *Computing*, 7(3–4):281–292, 1971.
39. Symbolic Computation Infrastructure for Europe project. `http://www.symbolic-computation.org/`.
40. William Stein. *Sage: Open Source Mathematical Software (Version 2.10.2).* The Sage Group, 2008. `http://www.sagemath.org`.
41. Dylan Thurston and Henning Thielemann. Haskell Numeric Prelude program. `http://darcs.haskell.org/numericprelude/`.
42. Philip W. Trinder, Ed. Barry Jr., M. Kei Davis, Kevin Hammond, Sahalu B. Junaidu, Ulrike Klusik, Hans-Wolfgang Loidl, and Simon L. Peyton Jones. GpH: An Architecture–Independent Functional Language. In *Glasgow Functional Programming Workshop*, Pitlochry, Scotland, September 1998.
43. Guido van Rossum. *The Python Language Reference Manual.* Network Theory Ltd., 2006.
44. Joachim von zur Gathen and Jürgen Gerhard. *Modern Computer Algebra.* Cambridge University Press, second edition, 2003.
45. J. H. Wilkinson. *Rounding Errors in Algebraic Processes.* Prentice Hall, 1963.
46. J. H. Wilkinson. The perfidious polynomial. In G. H. Golub, editor, *Studies in Numerical Analysis*, volume 24, pages 1–28. Mathematical Association of America, Washington, D. C., 1984.
47. Stephen Wolfram. *Mathematica: a system for doing mathematics by computer.* Wolfram Research, Inc., 1991.