What Happened to Languages for Symbolic Mathematical Computation?

Stephen M. Watt
Ontario Research Centre for Computer Algebra
University of Western Ontario
London Ontario, Canada N6A 5B7
watt@csd.uwo.ca

Abstract

While the state of the art is relatively sophisticated in programming language support for computer algebra, there has been less development in programming language support for symbolic computation over the past two decades. We summarize certain advances in programming languages for computer algebra and propose a set of directions and challenges for programming languages for symbolic computation.

1 Introduction

Digital computers have been used to solve symbolic mathematical problems now for more than half a century. While early work considered algebraic expressions and the operations of differential and integral calculus (*e.g.* [11, 17, 27]), now the full range of mathematics is considered from an algorithmic point of view. Also, where early work used computers to perform the same basic algorithms as used by people at elementary levels, now highly sophisticated techniques are used in mathematical algorithms, in mathematical experimentation and in automated proof.

There have been some remarkable advances, both theoretical and practical, extending the scope of problems that computer algebra can treat. Moreover, there are commercial and research systems that have brought computer algebra into the mainstream, where it may be used by anyone who deals with mathematical formulæ. The state of the art in symbolic algorithms for linear, polynomial and differential systems has evolved and continues to evolve rapidly. The ranks of researchers working in the area have grown considerably, now with many mathematicians contributing to the main algorithmic advances. It is now sometimes feasible to treat large problems exactly by symbolic means where approximate numerical methods fail.

What is wrong with this rosy state of affairs? The successes of symbolic mathematical computation have also been, to a certain extent, its undoing.

As algorithms in some areas have been developed, problems have been re-cast using the specialized languages of those areas. This has moved the focus *away from general techniques* for doing mathematics by computer, and toward the quest for *better algorithms in central areas*. Our focus on these areas has led to the exclusion of others and, moreover, to the tendency to see all problems only in these terms. To give just one example, it has been common practice for computer algebra systems to provide formal antiderivatives where integrals have been requested. Because system developers became so used to thinking in terms of differential algebra, they would miss the fact that (or even argue that it was correct that), as functions, the "integrals" could have spurious discontinuities and jumps.

While it is important to pursue technical development on the algebraic algorithms for core domains, there are embarrassing lacunae in the repertoire of what computer algebra can do. To give a simple example, although we may now factor multivariate polynomials of high degree in many variables over algebraic function fields, computer algebra systems cannot presently factor the simple expression $d^2 - 2^{n^2+n}$, which is a difference of squares for any integer n. This is not a contrived problem: perhaps every reader has had to simplify similar expressions when analyzing algorithms.

Corresponding to the nearly exclusive focus on core problem domains, there has been diminished attention on how programming languages can best support symbolic mathematical computation. Indeed, where there used to be a host of programming languages specializing in symbolic mathematical computation, there are now only a few. Moreover, there seems to be little current attention as to how they can be made more effective. To a first approximation, today we no longer have languages *for* symbolic mathematical computation. Instead we have general purpose languages that happen to be *applied to* symbolic mathematical computation.

Many ideas that later find their way into main-stream programming languages have appeared first in programming languages for mathematical computation. These ideas include programming with algebraic expressions, the use of arrays, arbitrary precision integers, automatic memory management and dependent types. We argue that there is still a lot left to learn: By looking harder at the problem domain of symbolic mathematical computation we can develop new language ideas that will first help symbolic mathematical computation, and then perhaps also be more generally applicable. This note is intended to encourage discussion on this topic.

We start with a discussion of the relation between computer algebra and symbolic computation. While this gives a computer algebra bias to the paper, we find it to be a useful example of the solution changing the problem. We then give a short summary of our own biases and outline why we think that programming language technology for computer algebra is in a useful state. We then consider the state programming languages for symbolic computation and identify a host of issues where we believe that language support could be both interesting and useful.

2 Computer algebra and symbolic computation

The term "symbolic computation" has different meanings to different audiences. Depending on the setting, it can mean: any computation that is not primarily arithmetical, any computation involving text, any computation on trees with free and bound variables, or computation involving codes over a set of symbols. In fact the first volume of the *Journal of Lisp and Symbolic Computation* (Kluwer 1991) and the first volume of the *Journal of Symbolic Computation* (Academic Press 1985) have almost disjoint subject matter. For our purposes, "symbolic computation" or "symbolic mathematical computation" will relate to mathematical computation with expressions, exact quantities or approximate quantities involving variables or indeterminates. We could adopt a broader or narrower definition, but this would not change the main points we wish to make.

In the early days of symbolic mathematical computing, much of what was done could be described as expression manipulation. This point of view is nicely captured in the survey by Jean Sammet [25]. For several years there was varying terminology involving combinations of "symbolic", "algebraic", "computation" and "manipulation" as well as the term "computer algebra." During this naming uncertainty the field itself was changing: First, broader classes of problems were addressed and specialized communities grew up around particular areas. Second, within many of these areas, problems were solved in specific algebraic structures rather than in terms of general expressions. Thus, to a certain extent, part of symbolic mathematical computing grew into the name "computer algebra."

Many working in the area of computer algebra use the terms "computer algebra" and "symbolic computation" synonymously. This author, however, finds it very useful to make a distinction between the two, even when working on the same types of objects. By "computer algebra," I mean the treatment of mathematical objects as values in some algebraic domain, for example performing ring operations on polynomials. In this view, $x^2 - 1$ and $(x - 1) \times (x + 1)$ are the same. By "symbolic computation," I mean working with terms over some set of symbols. In this view, the two expressions $x^2 - 1$ and $(x - 1) \times (x + 1)$ are different. Computing a gcd is computer algebra; completing a square is symbolic computation.

It is difficult in computer algebra to treat problems with unknown values (*e.g.* with a matrix with unknown size, a polynomial of unknown degree or with coefficients in a field of unknown characteristic). It is difficult in symbolic computation to use any but the most straightforward mathematical algorithms before falling back on general term re-writing. We must explore what can be done to bridge the gap between these two views. We can make computer algebra *more symbolic*, providing effective algorithms for a broader class of mathematical expressions. We can likewise make symbolic computation *more algebraic*, by restricting classes of admissible expressions, for example using typed terms.

As an example of this rapprochement, the papers [14, 33, 34] begin a discussion of the relationship between the arithmetic view of computer algebra and the term-rewriting view of symbolic computation, concentrating on polynomials of

unknown degree. The first objective is to formalize the notion of symbolic polynomials. The approach has been to take symbolic polynomials as expressions in finite terms of ring operations, allowing variable and coefficient exponents to be multivariate integer-valued polynomials, e.g. $x^{n^4-6n^3+11n^2-6(n+2m-3)} \_ 1000000^m$. This gives a group-ring structure, which, for suitable coefficient rings, can be shown to be a unique factorization domain. Factorizations in this UFD are uniform factorizations under evaluation of the exponent variables.

The cited papers begin to explore two approaches for algorithms to compute greatest common divisors, factorizations, *etc.* The first is to use the algebraic independence of x_i, $x_i^{n_1}$, $x_i^{{n_1}^2}$, $x_i^{n_1 n_2}$, *etc.* By introducing a number of new variables, we reduce the problem to one on usual multivariate polynomials. Avoiding fixed divisors of the exponent polynomials requires expressing them in a binomial basis. This, however, makes exponent polynomials dense, leading to doubly exponential algorithms. The second approach is to use an evaluation/interpolation scheme on the exponent variables. This has combinatorial problems when the polynomials have a limited set of distinct coefficients. Careful use of evaluation, and solving equations in symmetric polynomials, seems to point to a direction that avoids exponential combinations. Early experiments seem to show that this approach is amenable to the use of sparse interpolation techniques in the exponents. A simple application of Baker's transcendence theory allows us to treat logarithms of primes as algebraically independent so we can handle polynomial exponents on integer coefficients.

A number of authors have extended the domain of computer algebra in particular directions, *e.g.*, in the areas of mathematical theory exploration [1] and quantifier elimination [29]. More closely related to the current discussion are recent work on parametric [37] and uniform [23] Gröbner bases. Part of the motivation for the current direction for algorithms on symbolic polynomials comes from earlier work that extended the domain of polynomials to supersparse polynomials [12] and the ring of exponential polynomials [9]. A novel current direction is the algebraic treatment of elision ($\cdots$) in symbolic matrices [26]. There are isolated examples of early work on simplifying matrix-vector expressions [28] and the relation of symbolic computing to computer algebra [4].

3 Prejudices

For context, I should say that I have been involved in the design of several languages used in computer algebra, including Maple, Axiom, Aldor, OpenMath and MathML. The first three are programming languages, or systems with programming languages, and the last two are data languages.

Both Maple and the Axiom interpreter language try to provide a relatively simple to use language designed for scripting and interactive use. The Maple system uses its scripting language for most algebraic library development, while Axiom provides a distinct (but related) language for this purpose.

The Axiom library programming language had its initial version described in 1981 [10]. It provided an early implementation of qualified parametric poly-

morphism, which has decades later been adopted as an essential technique in the main-stream programming world.

Aldor [8, 31] extended the ideas of the Axiom programming language, adopting dependent types as its foundation and reconstructing Axiom's categories and domains, as well as object-oriented constructs from them. The use of *post-facto extensions* foreshadowed the separation of concerns by aspect-oriented programming, as reported in [35]. In addition, Aldor's use of abstract iteration [36] was an early example of the renaissance of control-based iterators (with `yield`), pioneered by CLU and Alphard, and now appearing in limited forms in languages such as Ruby and `C#`.

4 Language support for computer algebra

Computer algebra libraries tend to require "programming in the large," with precisely-related composable libraries, while at the same time requiring access to efficient arithmetic. In my opinion, the necessary programming language support for computer algebra is now quite mature and very good. Quite reasonable computer algebra libraries can be obtained using

- automated memory management,
- access to machine-level arithmetic and bit operations,
- parametric polymorphism, preferably with qualified parameters.

In some cases, mechanisms for very good support is available, *e.g.* to specify precise relationships among type parameters (Aldor), or to efficiently compile special cases of templates (`C++`).

Computer algebra is one of the few domains that provide rich and complex relationships among components that are at the same time well-defined. We therefore sometimes see programming language issues arising in area of computer algebra before they are seen in more popular contexts. We continue to see this today. Some of our recent work has included techniques for memory management [3, 30], performance analysis of generics [5, 6, 7], optimization of iterators and generics [5, 32, 36], and interoperability in heterogeneous environments [18, 19, 20, 21, 30, 38].

I see the following as interesting programming language problems whose investigation will benefit computer algebra:

(1) how to improve the efficiency of deeply nested generic types (templates, domain towers),

(2) techniques for efficient use of parametric polymorphism in multi-language environments,

(3) language support for objects that become read-only after an initial, extended construction phase,

(4) to use type categories to specify machine characteristics for re-configurable high-performance codes,

(5) a framework in which program-defined generic (parameterized) type constructors can form dependent types, e.g. `HashTable(k: Key, Entry(k))`, analogous to `(a: A) -> B(a)`,

(6) the use of *post facto* functor extensions as an alternative to open classes and aspect-oriented programming,

We have made some progress on some of these items: (1) and (2) have been the subject of PhD theses of two recent students. Some preliminary investigations on (3) have been undertaken together with a current postdoc. The topic (4) has been the subject of a collaboration with a group at U. Edinburgh that uses Aldor for Quantum Chromodynamics computations. The question (5) is motivated by the desire to treat all type constructors on an equal footing, an idea which has returned a lot of benefit in Aldor. There the mapping and product type constructors are still special only inasmuch as they provide dependent types. A model for generic dependency would allow mathematical programs to represent mappings either as functions, tables or other structures and allow all of these alternatives equal expressive power. Although the ideas behind (6) date back to work at IBM research in the early 90s, the idea of post facto extension of functors seems to just now be coming of age in the programming language world.

For problem (1), the Stepanov benchmarks provide a very simple measure for C++, which we have generalized to our SciGMark benchmark. Post-facto extensions are closely related to the notions of Aspect-Oriented Programming [13] and open classes [16], however they seem to provide more structure than the first and more opportunity for optimization than the second. Dependent types have for the longest time remained a secret of theorem provers and boutique theoretical languages, despite their early prominent occurrence AUTOMATH in the 1960s. The Aldor community has found them extremely useful. A limited form of dependency has been popularized by F-bounded polymorphism [2] and, more recently, dependent types have been receiving increased attention [15, 22].

5 Language support for symbolic computation

In contrast to the ample language support for computer algebra, there has been relatively little fundamental advance in language support for symbolic mathematical computation. In fact, with so many of the main advances being in algebraic algorithms, our software for symbolic mathematical computation have become popularly known as "Computer Algebra Systems."

Most computer algebra systems do in fact provide some level of support for expression manipulation. This typically includes expression traversal, expression reorganization into different forms (e.g. Horner form *vs* expanded form *vs* factored form), substitution, evaluation and various simplifications. Macsyma was an early system providing very good support in this area, and Mathematica is a more modern example of a system with good support in this area.

Despite this level of support for expression manipulation, I would suggest that programming language support for symbolic computation is still at an early stage. Some of the language directions that could fruitfully be explored include:

(1) better support for typed terms, *i.e.* variables that admit substitutions of only certain types and typed function symbols,

(2) construction of domains of symbolic terms (initial objects) by functorial operations on categories of concrete domains,

(3) smooth and extensible inter operation of program expressions on concrete domains and typed symbolic expressions as data,

(4) anti-unification modulo equational theories, use of adjoint functors for expression simplification, for example as in [24],

(5) use of empirical measures on expression spaces to define preferred forms of expressions under simplification,

(6) constructing expression transformations automatically from re-ordering of composed algebraic domain constructors ,

(7) generalizing the functional programming techniques of monads or arrows to move algorithms over commutative diagrams,

(8) more robust support for symbolic expressions on particular domains, including vector algebra and algebras of structured matrices,

(9) tools for encoding and using results from universal algebra,

(10) tools for better working with rule-sets, *e.g.* to determine noetherianity or divergence in certain settings,

(11) well-defined interfaces between automated theorem provers and computer algebra systems,

(12) technical matters, such as better support for hygienic treatment of free and bound variables in expressions.

These are a few of the immediate directions where symbolic computation could be better supported by linguistic mechanisms. It is a nice problem in language design to take full advantage of data/program duality so that meaningful composition of mathematical expressions can be constructed from composition of the corresponding programs. This is a fundamental area to get right if our symbolic mathematics systems are to be able to scale up. The "wild west" days of computer algebra systems working with ill-defined classes of expressions must be put behind us.

6 Conclusions

We have argued that the "computer algebra" and "symbolic mathematical computation" are two very different things and that, while computer algebra has been flourishing, symbolic computation on the same domain has been languishing. This is true both at the level of mathematical algorithms and in programming language support.

We have argued that the gap between these two areas must be bridged in order to have more useful systems for mechanized mathematics. These bridges can occur at the mathematical level, both by algebratizing the treatment of symbolic expressions and by developing algorithms for broader classes of algebraic objects. We gave the treatment of polynomials of symbolic degree as an example. These bridges should also occur at the programming language level, and we have presented a number of directions in which better language support could make symbolic computing richer.

What has happened to languages for symbolic mathematical computation? At least to this author, it appears that after the initial developments there have been many avenues left unexplored. Building sophisticated mathematical software that works, and that scales, is a difficult problem. If we are to succeed at building modular, extensible symbolic mathematics systems, then we should think harder about what we should ask of our programming languages.

References

[1] B. Buchberger. Algorithm-supported mathematical theory exploration: A personal view and strategy. In *Proc AISC*, pages 236–250. Springer, LNAI 3249, 2004.

[2] Peter Canning, William Cook, Walter Hill, and Walter Olthoff. F-Bounded Polymorphism for Object Oriented Programming. In *Proc. FPCA*, pages 273–280. ACM, 1989.

[3] Yannis Chicha and Stephen M. Watt. A localized tracing scheme applied to garbage collection. In *Proc. The Fourth Asian Symposium on Programming Languages and Systems*, pages 323–339. Springer Verlag LNCS 4279, 2006.

[4] J.H. Davenport and Ch. Faure. The "unknown" in computer algebra. *Programmirovanie*, 1:4–10, 1994.

[5] Laurentiu Dragan and Stephen Watt. Parametric polymorphism optimization for deeply nested types in computer algebra. In *Proc. Maple Conference 2005*, pages 243–259. Maplesoft, 2005.

[6] Laurentiu Dragan and Stephen M. Watt. Performance analysis of generics for scientific computing. In *Proc. 7th Internatioanl Symposium on Symbolic and Numeric Algorithms in Scientific Computing*, pages 93–100. IEEE Press, 2005.

[7] Laurentiu Dragan and Stephen M. Watt. On the performance of parametric polymorphism in maple. In *Proc. Maple Conference 2006*, pages 35–42. Maplesoft, 2006.

[8] S.M. Watt *et al. Aldor User Guide*. Aldor.org, 2003.

[9] C.W. Henson, L. Rubel, and M. Singer. Algebraic properties of the ring of general exponential polynomials. *Complex Variables Theory and Application*, 13:1–20, 1989.

[10] R. Jenks and B. Trager. A language for computational algebra. In *Proc. 1981 ACM Symposium on Symbolic and Algebraic Computation*, pages 6–13. ACM Press, 1981.

[11] H. G. Kahrimanian. *Analytical differntation by a digital computer*. MA Thesis, Temple University, Philadelphia PA, 1953.

[12] E. Kaltofen and P. Koiran. On the complexity of factoring bivariate supersparse (lacunary) polynomials. In *Proc. ISSAC'05*, pages 208–215. ACM, 2005.

[13] G. Kiczales, J. Lamping, and A. Mendhekar *et al*. Aspect-oriented programming. In *Proc. ECOOP*, pages 220–242. Springer LNCS 1241, 1997.

[14] Matthew Malenfant and Stephen M. Watt. Sparse exponents in symbolic polynomials. In *Proc. Symposium on Algebraic Geometry and Its Applications: in honor of the 60th birthday of Gilles Lachaud*. (to appear), 2007.

[15] James McKinna. Why dependent types matter. In *Proc. POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–1. ACM, 2006. Extended draft by Altenkirch, McBride, McKinna available on-line www.dcs.st-andrews.ac.uk.

[16] T. Millstein and C. Chambers. Modular statically typed multimethods. In *Proc. ECOOP*, pages 279–303. Sprinter LNCS 1628, 1999.

[17] J. Nolan. *Analytical differentiation on a digital computer*. MA Thesis, Math Dept MIT, Cambridge MA, 1953.

[18] C. Oancea and S.M. Watt. A framework for using aldor libraries with maple. In *Actas de los Encuentros de Algebra Computacional y Aplicaciones (EACA) 2004*, pages 219–224. Universitad de Cantabria, ISBN 84-688-6988-04, 2004.

[19] C. Oancea and S.M. Watt. Domains and expressions: An interface between two approaches to computer algebra. In *Proc. International Symposium on Symbolic and Algebraic Computation*, pages 261–268. ACM Press, 2005.

[20] Cosmin Oancea and Stephen M. Watt. Parametric polymorphism for software component architectures. In *Proc. 20th Annual ACM Conference on Object Oriented Programming, Systems, Languages and Applications*, pages 147–166. ACM Press, 2005.

[21] Cosmin E. Oancea and Stephen M. Watt. Generic library extension in a heterogeneous environment. In *Proc. Library Centric Software Design*, 2006.

[22] M. Odersky, V. Cremet, C. Röcl, and M. Zenger. A nominal theory of objects with dependent types. In *Proc. ECCOP'03*, pages 201–224. Sprinter LNCS 2743, 2003.

[23] W. Pan and D. Wang. Uniform Gröbner bases for ideals generated by polynomials with parametric exponents. In *Proc. ISSAC'06*, pages 269–276. ACM, 2006.

[24] L. Pottier. Generalisation de termes en theorie equationnelle. Cas associatif-commutatif. Technical Report RR-1056, INRIA, 1989.

[25] J. Sammet. A survey of formula manipulation. *C. ACM*, 9(8):555–569, 1966.

[26] A. Sexton and V. Sorge. Abstract matrices in symbolic computation. In *Proc. ISSAC'06*, pages 318–325. ACM, 2006.

[27] J. R. Slagle. A hueristic program that solves symbolic integration problems in freshman calculus. *J. ACM*, 10(4):507–520, 1963.

[28] D. R. Stoutemyer. Symbolic computer vector analysis. *Computers and Mathematics with Applications*, 5(1):1–9, 1979.

[29] Th. Sturm. New domains for applied quantifier elimination. In *Computer Algebra in Scientific Computing, CASC 2006*, pages 295–301. Springer, LNCS 4194, 2006.

[30] S.M. Watt. A study in the integration of computer algebra systems: Memory management in a maple-aldor environment. In *Proc. International Congress of Mathematical Software*, pages 405–410. World Scientific 2002, 2002.

[31] S.M. Watt. Aldor. In V. Weispfenning J. Grabmeier, E. Kaltofen, editor, *Handbook of Computer Algebra*, chapter 4.1.2, pages 265–270. Springer Verlag, Heidelberg 2003 ISBN 3-540-65466-6, 2003.

[32] S.M. Watt. Optimizing compilation for symbolic-numeric computing. In *Proc. 6th International Symposium on Symbolic and Numeric Algorithms for Scientific Computation*, pages 18–. MITRON Press ISBN 973-661-441-7, 2004.

[33] Stephen M. Watt. Algorithms for symbolic polynomials. In *Proc. 9th International Workshop on Computer Algebra in Scientific Computing*, pages 302–. Springer Verlag LNCS 4194, 2006.

[34] Stephen M. Watt. Making computer algebra more symbolic. In *Proc. Transgressive Computing 2006: A conference in honor or Jean Della Dora*, pages 43–49, 2006.

[35] Stephen M. Watt. Post facto type extension for mathematical programming. In *Proc. ACM SIGSAM/SIGSOFT Workshop on Domain-Specific Aspect Languages*, 2006.

[36] Stephen M. Watt. A technique for generic iteration and its optimization. In *Proc. ACM SIGPLAN Workshop on Generic Programming 2006*, pages 76–86. ACM, 2006.

[37] V. Weispfenning. Gröbner bases for binomials with parametric exponents. Technical report, Universität Passau, Germany, 2004.

[38] C. Oancea Y. Chicha, M. Lloyd and S.M. Watt. Parametric polymorphism for computer algebra software components. In *Proc. 6th International Symposium on Symbolic and Numeric Algorithms for Scientific Computation*, pages 119–130. MITRON Press ISBN 973-661-441-7, 2004.