On the Performance of Parametric Polymorphism in Maple

Laurentiu Dragan   Stephen M. Watt

Ontario Research Centre for Computer Algebra
University of Western Ontario
London, Ontario, Canada N6A 5B7

{ldragan,watt}@orcca.on.ca

Abstract

With the introduction of support for generics in mainstream programming languages, we see an renewed interest in writing generic code. Maple offers the possibility to write generic code using module-producing functions. There is usually a performance cost associated with the use of generics, and this paper analyzes the cost of using these in Maple.

We created sets of tests for different styles of generics and compared their performance with specialized Maple code. The first set of tests implemented generics in the style of parameterized abstract data types, with Maple modules providing functions that operated on data values. The second set of tests implemented generics in the style of parameterized object-oriented classes, where each data value was embodied in its own module object. The test functions were based on the popular SciMark benchmark for scientific computation in Java. The results show a small degradation of performance when using generics modeled on abstract data types and high performance degradation when using an object-oriented approach.

Keywords. generics, parametric polymorphism, performance, Maple.

1 Introduction

Scientific algorithms are well suited to a generic style of programming due to their natural mathematical structure. The feasibility of generic code for scientific computing has been investigated in [4], [5]. Good examples of the usefulness of the generic libraries are provided by the C++ standard template library (STL) and the Boost libraries [6]. There are many computer algebra libraries using parametric polymorphism for generics: the NTL library for number theory [7], the LinBox library for symbolic linear algebra [8], the Sum-IT library for differential operators [9] and the Triade library for triangular sets [10], to mention just a few.

We believe that programming generically using parametric polymorphism is an important abstraction technique that can simplify code, allow more reuse and improve maintainability. Naive implementations of generic code typically do not perform as well as specialized code, but optimizers can make significant improvements and minimize the extra cost.

One of the prerequisites for wider adoption of generic code is a clear understanding of its cost. To address this question in a wider setting we have developed "SciGMark" a benchmark for generics in scientific computing. This is an extension of the well-known SciMark benchmark [3] to use generics. This benchmark suite contains various language implementations of a number of mathematical test problems. The first version of SciGMark contains the problems from SciMark for Java, C++, C# and Aldor in both specialized and generic form.

As described in the "Advanced Programming Guide" [11], Maple offers support for generic programming using modules produced by functions. It is the purpose of this paper to investigate the cost this incurs and the potential performance that may be gained through optimization. To conduct this analysis, we re-implemented our existing SciGMark test suite in Maple to compare the performance of specialized and generic programs. We produced two versions, corresponding to the object-oriented and the abstract data type models of computation. The object-oriented tests pair data with operations and the abstract data type tests separate the data from the operations. We show the results obtained for this benchmark, comparing the generic object-oriented, the generic abstract data type and the specialized versions. The environment provided by Maple is interpreted and, as a result, the overall performance of code is slower than the performance for the other general purpose languages used in our SciGMark test. It is not our intention, however, to compare Maple against Java, C++, C# and Aldor. Our goal, rather, is to investigate the implications of writing generic code in Maple.

This remainder of the paper is organized as follows: Section 2 is a brief description of SciGMark with its tests. Section 3 presents the implementation details of SciGMark in Maple. Section 4 shows the results obtained. Finally, Section 5 presents the conclusions.

2 SciGMark, a Generic SciMark

SciMark [3] is a Java benchmark for scientific and numerical computing and has been ported to C++ and C#. It measures several computational kernels and determines a composite score in approximate MFlops (Millions of FLoating point Operations Per Second). The computational kernels include the following:

> **Fast Fourier transform (FFT)** performs a one-dimensional forward transform of 4K complex numbers. This kernel exercises complex arithmetic, shuffling, non-constant memory references and trigonometric functions.

Jacobi successive over-relaxation (SOR) on a 100x100 grid exercises typical access patterns in finite difference applications, for example, solving Laplace's equation in 2D with Dirichlet boundary conditions. The algorithm exercises basic "grid averaging" memory patterns, where each A(i,j) is assigned an average weighting of its four nearest neighbors.

Monte Carlo integration approximates the value of π by computing the integral of the quarter circle $y = \sqrt{1 - x^2}$ on $[0, 1]$. It chooses random points within the unit square and computes the ratio of those within the circle. The algorithm exercises random-number generators, synchronized function calls, and function inlining.

Sparse matrix multiply uses an unstructured sparse matrix stored in compressed-row format with a prescribed sparsity structure. This kernel exercises indirection addressing and non-regular memory references.

Dense LU matrix factorization computes the LU factorization of a dense 100x100 matrix using partial pivoting. This exercises linear algebra kernels (BLAS) and dense matrix operations. The algorithm is the right-looking version of LU with rank-1 updates.

We have developed SciGMark [1] as an extension to SciMark using *generic* versions of the tests present in SciMark. This allows us to measure the difference in performance between generic and specialized code. The individual kernels are re-written to operate over a generic numerical type supporting the ring operations ($+, -, \times, /$, zero, one). The current version implements a wrapper for double precision floating point numbers and this is used to instantiate the tests. The tests may also be run in single or multiple precision, or using exact arithmetic by instantiating them with another numerical ring.

3 A Maple Version of SciGMark

Parametric polymorphism can be achieved in Maple using module-producing functions. The basic mechanism is to write a function that takes one or more modules as parameters and produces a module as its result. The module produced uses operations from the parameter modules to provide abstract algorithms in a generic form. The following is a simple example:

```
MyGenericType := proc(R)
    module ()
        export f, g;
        # Here f and g can use u and v from R.
        f := proc(a, b) foo(R:-u(a), R:-v(b)) end;
        g := proc(a, b) goo(R:-u(a), R:-v(b)) end;
    end module
end proc:
```

We investigated two ways of using this basic idea to provide generics:

- The first way — the "object oriented" (OO) approach — represented each value as a module. This module had a number of components, including fields (locals or exports) for the data and for the operations supported. Each value would be represented by its own constructed module.
- The second way — the "abstract data type" (ADT) approach — represented each value as some data object, manipulated by operations from some module. One module was shared by all values belonging to each type, and the module provided operations only. The data was free-standing.

We produced two Maple versions of SciGMark: one for each of these approaches. In both the OO and ADT versions, the SciGMark kernels use numerical operations from a generic parameter type. For the concrete instantiation of this parameter, we created the module "`DoubleRing`" as a wrapper for floating point.

The code for the OO version of `DoubleRing` is as follows:

```
DoubleRing := proc(val::float)
    local Me;
    Me := module()
        export v, a, s, m, d, gt, zero, one,
               coerce, absolute, sine, sqroot;
        v  := val;
        a  := (b) -> DoubleRing(Me:-v + b:-v);
        s  := (b) -> DoubleRing(Me:-v - b:-v);
        m  := (b) -> DoubleRing(Me:-v * b:-v);
        d  := (b) -> DoubleRing(Me:-v / b:-v);
        gt := (b) -> Me:-v > b:-v;
        zero    := () -> DoubleRing(0.0);
        one     := () -> DoubleRing(1.0);
        coerce  := () -> Me:-v;
        absolute:= () -> DoubleRing(abs(v));
        sine    := () -> DoubleRing(sin(v));
        sqroot  := () -> DoubleRing(sqrt(v));
    end module:
    return Me;
end proc:
```

This version simulates the object-oriented model by storing the value and the operations in a module. Each call to `DoubleRing` produces a new module that stores its own value. The exports `a`, `s`, `m` and `d` correspond to addition, subtraction, multiplication and division. We chose these names, rather than `` `+` ``, `` `-` ``, `` `*` `` and `` `/` ``, since Maple's support for overloading basic operations is rather awkward and we were not producing a piece of code for general distribution. The last two functions, `sine` and `sqroot`, are used only by the FFT kernel to replace complex operations and to test the correctness of the results.

The code for the ADT version of `DoubleRing` is as follows:

```
DoubleRing := module()
    export a, s, m, d, zero, one,
           coerce, absolute, sine, gt, sqroot;
    a  := (a, b) -> a + b;
    s  := (a, b) -> a - b;
    m  := (a, b) -> a * b;
    d  := (a, b) -> a / b;
    gt := (a, b) -> a > b;
    zero := () -> 0.0;
    one  := () -> 1.0;
    coerce   := (a::float) -> a;
    absolute := (a) -> abs(a);
    sine     := (a) -> sin(a);
    sqroot   := (a) -> sqrt(a);
end module:
```

It can be seen that this approach does not store the data; it provides only the operations. As a convention, one must coerce the float type to the representation used by the module. In this case the representation used is exactly float (as can be seen from the coerce function). The DoubleRing module is created only once when the module for each kernel is created.

Each SciGMark kernel exports an implementation of its algorithm and a function to compute the estimated floating point instruction rate. Each of the kernels is parametrized by a module, `R`, that abstracts the numerical type. An example of this structure is as follows:

```
gFFT := proc(R)

    module()
        export num_flops, transform, inverse;
        local transform_internal, bitreverse;

        num_flops := ...;
        transform := proc(data::array) ... end proc;
        inverse := proc(data::array) ... end proc;
        transform_internal := proc(data, direction) ... end proc;
        bitreverse := proc(data::array) ... end proc;
    end module:

end proc:
```

The high-level structure of the implementation is the same in both the OO and ADT generic cases. The detailed implementations of the functions in the module are different, however. An example of the same piece of code in all three cases is shown in Table 1. One can see that the specialized version makes use of

Model	Code
Specialized	`x*x + y*y`
Object-Oriented	`(x:-m(x):-a(y:-m(y))):-coerce()`
Abstract Data Type	`R:-coerce(R:-a(R:-m(x,x), R:-m(y,y)))`

Table 1: Differences in implementation of specialized and generic code

the built-in Maple operations. In this case, the values use Maple's native floating point representation. The other two versions make use of exported operations from `R`, which in our case is given by `DoubleRing`. The object-oriented model uses a module instance to obtain the operations associated with the data. One can see that in the object-oriented model the variables are themselves modules and are used to find the operations. On the other hand, the abstract data type model uses a module for the operations that is not connected to the data in any explicit way. In the abstract data model, the parameter passed in to the kernel module is the same for all operations on all data.

We tested the kernels described in Section 2. These were implemented in the same way in Maple as in other languages. In particular, we did not make use of Maple's own arithmetic to treat complex values and matrices as single objects. By doing this, and by taking tests where the parameter values were relatively light weight (floating point numbers), we hoped to expose the worst case performance of generics.

4 Results

The results of running SciGMark in Maple 10 are presented in Table 2. The benchmark was run on a Pentium 4 processor with 3.2 GHz, 1MB cache and 2 GB RAM. The operating system used was Linux, Fedora Core 4.

The results show that abstract data type model is very close in performance to the specialized version. The ratio between abstract data type and specialized versions is roughly 1.3. This means there is not strong justification, based on performance alone, to avoid writing generic algorithms in Maple. We should point out two situations, however, that require special consideration: The first is that with several nested levels of generic construction the compounding of the performance penalty may become significant. The second consideration is that some Maple procedures obtain their performance from an evaluation mode, `evalhf`, that treats hardware floats specially. Our investigation assumes `evalhf` is not being used.

The last column of Table 2 shows the results for the object-oriented model. This model tries to simulate as closely as possible the original SciGMark test, given the language features offered by Maple. This model constructs many modules during the benchmark, leading to a significant performance degradation. The ratio between object-oriented and specialized versions is 9.9; that is, the generic OO code is about one order of magnitude slower than the specialized code. This shows that this approach to writing generic code should be avoided

Test	Specialized	Abstract Data Type	Object Oriented
Fast Fourier Transform	0.123	0.088	0.0103
Successive Over Relaxation	0.243	0.166	0.0167
Monte Carlo	0.092	0.069	0.0165
Sparse Matrix Multiplication	0.045	0.041	0.0129
LU factorization	0.162	0.131	0.0111
Composite	0.133	0.099	0.0135
Ratio	1.0	1.3	9.9

Table 2: SciGMark MFlops in Maple 10

in Maple. If generic object-oriented code is truly required for some application, it would be worthwhile to explicitly separate the instance-specific data values from a shared-method module. Then values would be composite objects (e.g. lists) with one component being the shared module.

The performance penalty for generic code should not discourage writing of generic code, but rather encourage compiler writers to think harder about optimizing generic code constructs. Generic code is useful, they provide a much needed code reuse that can simplify the libraries. An example of such optimization has been proposed by specializing the type according to the particular parameter used when constructing the type, as mentioned in [2].

5 Conclusion

There are certain benefits that a generic programming style provides, including improved modularity, improved maintainability and re-use, and decreased duplication. In a mathematical context, writing programs generically also helps programmers operate at a higher, more appropriate level of abstraction. With these potential benefits, it is important to understand whether there are opposing reasons that preclude use of this style. We have made a quantitative assessment of the performance impact of using a generic programming style in Maple.

We have found that writing generic code using parametric polymorphism and abstract data types does not introduce an excessive performance penalty in Maple. We believe that this is in part due to the fact that Maple is interpreted and there is little overall optimization. Even specialized code executes function calls for each operation. Carefully written generic code and code that is not excessively generic can do well in Maple environment. We suggest that it would be worthwhile to consider modifications to the Maple programming language to make generic programming easier.

Writing code in Maple that tries to simulate the sub-classing polymorphism provided by object-oriented languages such as Java, can be very expensive in Maple. The code written using this approach can be an order of magnitude slower compared to the specialized code.

References

[1] L. Dragan and S. M. Watt *Performance Analysis of Generics in Scientific Computing*, Proceedings of Seventh International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, Timisoara, Romania, IEEE Computer Society, 90–100, 2005

[2] L. Dragan and S. M. Watt *Parametric Polymorphism Optimization for Deeply Nested Types*, Proceedings of Maple Conference 2005, Waterloo, Canada, Maplesoft, 243–259, 2005

[3] R. Pozo and B. Miller, `SciMark2`, Natl. Inst. of Standards and Technology, `http://math.nist.gov/scimark2`

[4] J. Gerlach and J. Kneis, *Generic Programming for Scientific Computing in C++, Java, and C#.*, Advanced Parallel Programming Technologies, 5th International Workshop, 2834, Xiamen, China, Springer, 301–310, 2003

[5] T. L. Veldhuizen and M. E. Jernigan, *Will C++ be faster than Fortran?*, Proceedings of the 1st International Scientific Computing in Object-Oriented Parallel Environments (ISCOPE'97), Marina del Rey, California, Springer-Verlag, 1997

[6] D. Abrahams, *Boost C++ Libraries*, `http://www.boost.org/`, 2002

[7] V. Shoup, *NTL: A Library for Doing Number Theory*, `http://www.shoup.net/ntl/doc/tour.html`

[8] J.-G. Dumas, T. Gautier, M. Giesbrecht, P. Giorgi, B. Hovinen, E. Kaltofen, B. D. Saunders, W. J. Turner and G. Villard, *LinBox: A Generic Library For Exact Linear Algebra*, International Congress of Mathematical Software (ICMS 2002), Beijing, China, `http://citeseer.csail.mit.edu/511043.html`

[9] M. Bronstein, *SUM-IT: A Strongly-Typed Embeddable Computer Algebra Library*, Proceedings of DISCO'96, Karlsruhe, Germany, Springer LNCS 1128, 22–33, 1996

[10] M. Moreno Maza, *On Triangular Decompositions of Algebraic Varieties*, Méthodes Effectives en Géométrie Algébrique (MEGA 2000), Bath, UK, (`http://www.bath.ac.uk/~masdr/abst/`), 2000

[11] M. B. Monagan, K. O. Geddes, K. M. Heal, G. Labahn, S. M. Vorkoetter, J. McCarron and P. DeMarco, *Maple 10 – Advanced Programming Guide*, Maplesoft 2005