Type Specialization in Aldor

Laurentiu Dragan and Stephen M. Watt

Computer Science Department
The University of Western Ontario
London, Canada
{ldragan,watt}@csd.uwo.ca

Abstract. Computer algebra in scientific computation squarely faces the dilemma of natural mathematical expression versus efficiency. While higher-order programming constructs and parametric polymorphism provide a natural and expressive language for mathematical abstractions, they can come at a considerable cost. We investigate how deeply nested type constructions may be optimized to achieve performance similar to that of hand-tuned code written in lower-level languages.

1 Introduction

One of the challenges for computer algebra in scientific computation is to achieve high performance when both symbolic and numeric computation are required. Aldor has sought to do this without sacrificing the natural high-level expression of mathematical algorithms.

Aldor has been designed to provide a rich set of composable abstraction mechanisms so that mathematical programs can be written in a natural style. These mechanisms use dependent types and type-producing functions to allow programs to be written in their most general form, and then specialized. The Aldor compiler manages to achieve performance comparable to optimized C or Lisp for programs that perform numeric or symbolic computations, respectively.

Aldor's type-producing functions can be used in a similar manner to templates in C++ or "generics" in Java, but with more natural mathematical structure. Even though writing generic code is highly desirable from the code reusability perspective, programmers often avoid this approach for performance reasons. Deeply nested generic type constructions (which occur naturally in computer algebra) can hinder the performance of programs to an extent that entices some programmers to specialize code by hand. To avoid this, we propose an optimization that automatically specializes generic code based on the particular type instantiations used in a program. Our optimization specializes both the code and the data representation of the type, producing programs up to an order of magnitude faster.

Beyond its applicability to computer algebra, this work has implications for main-stream programming languages. Because programs in languages such as C++, Java and C# are only now starting to use templates extensively, the performance problems associated with deeply nested generic types have not been widely recognized in those settings.

The contributions of this paper are:

- a compiler optimization that specializes dynamic domain constructors when instantiations are known at compile time,
- an extension to the type specialization optimization that specializes also the data representation of the specialized type, and
- some numerical indication of the improvements that can be expected by an automatic process, and some results for a hand-crafted example.

The remainder of this paper is organized as follows: Section 2 gives a brief introduction to parametric polymorphism implementations and introduces the Aldor programming language and its compiler. Section 3 presents the problem presented by deeply nested type constructions. Section 4 describes the method of code specialization. Section 5 presents the approach used for data specialization. Section 6 presents some performance results and Section 7 concludes the paper.

2 Background

Approaches to parametric polymorphism: There are currently two approaches to implement parametric polymorphism, the *homogeneous* and *heterogeneous* approaches.

The *heterogeneous* approach constructs a special class for each different use of the type parameters. For example, with `vector` from the C++ standard template library, one can construct `vector<int>` or `vector<long>`. This generates two distinct classes that we may think of as: `vector_int` and `vector_long`. This is done by duplicating the code of the vector generic class and producing specialized forms. This is an effective approach from the time efficiency point of view. Two drawbacks of this method are the size of the code and that all parameters must be known statically. Another drawback is that constructors may not themselves be passed as run-time parameters.

The *homogeneous* approach uses the same generic class for every instance of the type parameters. This method is used in Java by erasing the type information, using the Object class in place of the specialized form, and by type casting whenever necessary. This method introduces little run-time overhead, but misses out on the optimizations possible in special cases. More importantly, the code size is not increased at all. For example, Java's `Vector<Integer>` is transformed to `Vector` containing `Object` class objects, and the compiler ensures that `Integer` objects are used as elements of the vector.

The Aldor Programming Language: Aldor was originally designed as an extension programming language for Axiom computer algebra system. It can be used more generally, but its affinity for computer algebra can be seen in its extensive algebra related libraries. One of the strengths of Aldor is that functions and types are first-class values. This way, functions and types can be computed and manipulated just as any other values. The type system in Aldor is organized on two levels, "*domains*" and "*categories*". Domains provide datatypes, and categories provide a sub-type lattice on the domains. More details about the Aldor programming language can be found elsewhere [1–3].

The Aldor Compiler: One of the main design goals of the Aldor programming language was to allow efficient implementation of symbolic and numeric mathematical programs. Aldor's compiler therefore implements many optimizations, including procedure inlining, data-structure elimination and control flow optimization. The Aldor compiler uses a platform-independent intermediate code, FOAM (First Order Abstract Machine), with efficient C and Lisp mappings.

After syntactic and semantic analysis, the compiler performs straightforward FOAM code generation, followed by a series of aggressive FOAM-to-FOAM optimizations, producing code that is close in performance to optimized C code.

3 Deeply Nested Type Constructions

We now turn our attention to the main problem, that of deeply nested type constructions. We define the *depth* of a type construction expression to be the number of generic constructors along the longest path from the root of the expression tree to any leaf. An example of a nested type construction is given by the following expression:

```
Set (Matrix (Poly (Complex (Fraction(Integer)))))
```

This expression forms the type of sets whose elements are matrices, the elements of which are polynomials with complex rational coefficients. This is a fully explicit nested type expression. Even though `Set`, `Matrix`, `Poly`, `Complex` and `Fraction` are parametric types, the resulting type is no longer parametric. This kind of fully explicit construction is not very common in Aldor algebra code. It is more usual to see constructions with a few levels, given explicitly, applied to a type parameter, and for this type parameter to be given a value at run time that is a construction of a few levels on another parameter, etc. The result is a deeply nested tower of constructions, built a few layers at a time. Similarly, with the expansion of macros and `typedef`s in C++, deeply nested type constructions frequently arise.

Continuing with our example above, let us imagine that one would call an operation from the `Set` domain to multiply each element by a constant. The set elements are matrices, which requires calling a function (to multiply a matrix by a constant) from the `Matrix` domain. Each element from the matrix is a polynomial which requires invoking the constant multiplication operation from the `Poly` domain, and so on. This operation requires many call frame activations and de-activations. This introduces an overhead that can be avoided by specializing the domain. After specializing the operations of the domain, it is usually possible to optimize the resulting operation further, starting with procedure inlining.

It is often the case that the leaves of type expressions are given by parameters, for example as with the parameter `R` in

```
MyConstruction(R: IntegralDomain): Algebra(R) == {
      s: Set (Matrix (Poly (Complex (Fraction(R))))) := ...
}
```

This leads us to consider the idea of constructing a specialized domain constructor:

$$\texttt{Set} \circ \texttt{Matrix} \circ \texttt{Poly} \circ \texttt{Complex} \circ \texttt{Fraction}$$

The functions from this domain constructor could be specialized using functions from the intermediate domain constructors, even if `R` is not known.

In Aldor, type constructors may themselves be dynamically bound so it is possible that we may not know one of the constructors in a deeply nested expression at compile-time. For example, we may have constructors such as:

$$\texttt{Set} \circ \texttt{Matrix} \circ \texttt{Poly} \circ \texttt{Complex} \circ X$$
$$X \circ \texttt{Matrix} \circ \texttt{Poly} \circ \texttt{Complex} \circ \texttt{Fraction}$$
$$\texttt{Set} \circ \texttt{Matrix} \circ X \circ \texttt{Complex} \circ \texttt{Fraction}$$

where X is an domain constructor unknown at compile-time. These cases must also be handled. For example, in the last line above, this means handling the specialized constructors $F = \texttt{Set} \circ \texttt{Matrix}$ and $G = \texttt{Complex} \circ \texttt{Fraction}$ to build $F \circ X \circ G$ at run-time.

4 Code Specialization

As mentioned previously, generic functions and type-constructing functions ("functors") in Aldor are implemented using the homogeneous approach. While this is very flexible, performance can suffer. The goal of present work was to use a mixture of homogeneous and heterogeneous implementations for deeply nested type expressions to improve efficiency. What makes this different from the analogous question in other programming languages is that in Aldor types are constructed dynamically and, as seen above, both the leaf types and the constructors themselves may be parameters unknown at compile time.

Some authors, e.g. [4], view partial evaluation as program specialization. Partial evaluators typically specialize whole programs rather than individual functions. For example, a partial evaluator may take program P and some of the inputs of that program, and produce a residual program R, that takes only the rest of the inputs of P and produces the same result as P, only faster. We take a similar approach for type specialization. We take a FOAM program in the representation used by the Aldor compiler and we specialize it according to some type instantiations.

In our case, we use a partial evaluator that not only specializes the domain constructor, but also specializes all the exports of that domain, effectively specializing the code of the type constructed by the functor. This creates operations of that specialized domain as monolithic operations that are more efficient. The overhead of the domain creation is more significant, but it does not happen very frequently. The main part of the speedup does not come from eliminating domain creation overhead. Rather it comes from from the optimization of the specialized functions.

Domain Specialization

1. Initialize the data structures.
2. Identify the domain declarations.
3. For each program,
   (a) If there is a domain constructor, generate a new specialized domain based on the domain constructor.
   (b) Replace the call to the generic domain constructor to a call to the specialized one.
   (c) In the specialized domain try to find the imports from other domains, and if found, modify the FOAM representation to help Aldor in-liner identify the called function.
4. Construct the FOAM code back from the data structures used by the tower optimizer.

Fig. 1. Algorithm to specialize domain code by FOAM-to-FOAM transformation.

Our method tries to create specialized forms of the functions of instantiated domains. These are then used to construct the specialized run-time domains. The specialized functions should in-line code from domains that are type parameters. This way the resulting domains will have functions that do not require expensive function dispatch when executing the code from an instance domain.

Aldor domains are run-time entities, and are therefore not always completely defined at compile-time. In such cases, a full optimization is not possible. However, even in these cases parts of the type can be defined statically as in the examples given in Section 3. In these cases, a partial specialization is still possible. The algorithm used to transform the FOAM code is presented in Figure 1.

The FOAM code corresponding to Aldor source comprises two parts: the declaration of the data and the programs that manipulate that data. The optimization performs a single scan of all the programs found in FOAM and looks for functors. For each functor found, the type information for the original domain is retrieved as saved by the type analysis phase or it is reconstructed. Then the code for the type expression (i.e. the domain construction and all its operands) is cloned. Once all the operations of the original domain have been duplicated, the resulting cloned domain is updated by scanning all the operations for imports from the domains used as parameters. Once a target program has been found, the caller marks its call site with the address of the callee. This way the usual inliner can find the target function to be expanded locally. The final decision, whether the function should actually be expanded locally, remains with the general purpose inliner. This approach avoids expanding all the functions and it relies on the rules of the usual inliner to expand only the worthwhile cases.

In many cases not all parts of a type expression are known at compile time. In this situation, the above procedure is applied to the parts that are known. The specialized types will preserve the signature of all the exported symbols, so they can be used instead of the original call without affecting the rest of the caller's code. Our specialization is done in pairs of two starting from the innermost to the outermost domain constructor.

Preliminary results, presented in [6], showed that most of the speedup is obtained by specializing the innermost domains. For example, in case of a deeply nested type `Dom4(Dom3(Dom2(Dom1)))`, most of the speedup is obtained by specializing `Dom2(Dom1)`. On the other hand, specializing `Dom4(Dom3(X))` will not produce as significant a speedup.

This optimization is restricted to those cases where the type is fully or partly known at compile time. For those types that are completely dynamically constructed at run-time, as is the case with some functions able to generate new types at run-time, this transformation is not applicable and a dynamic optimizer must be used.

5 Data Specialization

Another important optimization that can be performed on opaque domains is data representation specialization. We have found this can have a very significant performance impact. We see this already in other environments: Even though parametric polymorphism has been introduced to Java and C#, their performance is still not as good as specialized code. We now describe our data specialization and how we measured the performance improvement.

While trying to measure the performance, we searched for benchmarks to measure generic code performance. We found only Stepanov's abstraction "penalty" benchmark [9], which is rather simple. To obtain meaningful measurements we transformed a well-known benchmark for scientific computing (SciMark) to use generic code (SciGMark) [7]. While constructing SciGMark we experimented with different types of specializations and discovered that most of the speedup between the hand specialized code and generic code was achieved when the data representation was changed from heap allocated to stack allocated structures.

In the process of implementing SciGMark, we noticed that a considerable speedup was obtained from data representation specialization. This transformation is possible if the data representation is private, because access to data is always through accessor functions. This way there is no risk of access to data through aliases. This is indeed the case with the representation of Aldor domains.

The Aldor compiler already offers an optimization for data structure elimination, which tries to flatten records, eventually moving heap allocation to the stack or registers if the objects do not escape local scope. A similar escape analysis for Java was presented by Choi [8]. Our proposed specialization goes a step further by eliminating data structures not only in each function, but also across operations that belong to a specialized domain.

The idea behind this optimization is to incorporate the data structure associated with the inner domain into the data structure of the outer domain. Since the data representation is private to the domain, this change will be invisible to its clients. The passing or conversion of data will be handled automatically by the operations of the transformed domain. The rest of the program can remain unchanged.

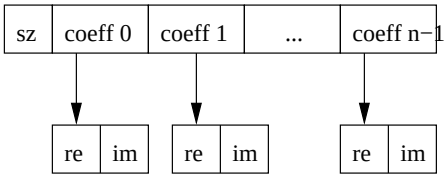

Fig. 2. Data rep. for polynomial with complex coefficients (before specialization.)

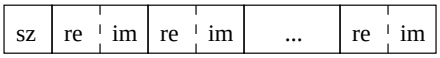

Fig. 3. Data rep. for polynomial with complex coefficients (after specialization.)

To illustrate how this works, we use a simple polynomial multiplication as example. One usual representation of a polynomial, using a dense representation, is as an array of coefficients values. A generic polynomial type constructor can accept different algebraic types for the coefficients. Suppose we use a complex number type as coefficients. In the same way, the complex type constructor can be generic, accepting a type for the real and imaginary parts. Suppose we take these to be integers. This is created in Aldor using `Poly(Complex(Integer))` and the data representation is an array similar to Figure 2. If the polynomial is specialized for such complex numbers, the resulting representation can be seen in Figure 3. This new representation has a much better memory locality, eliminates some indirections and more importantly, it eliminates the need to allocate and manage heap objects.

To illustrate how data representation specialization works, we created a small example that multiplies two polynomials that use a dense representation for the coefficients.

The implementation of `Ring` category and the `Complex` domain can be seen in Figure 4. The `Ring` category declares a type that has operations such as addition, multiplication, the ability to construct a neutral element and to print on screen. The `Complex` domain is an example of a domain that implements a `Ring` type, by providing implementations for all the operations declared in `Ring`.

In Figure 4, the Complex domain is not a parametric domain. In this case, the data representation used for the real and imaginary part is the domain `MachineInteger`. We take this simple approach for this example, because we shall perform the code and data specialization by hand, without any help from an automatic system. In this case the type of `Complex` is a `Ring` extended with some extra operations like `complex`, `real` and `imag`. The identifier `%` (percent) is the local name of the domain itself, The two macros `per` and `rep` are used to convert between the `%` type (the abstract type) and the `Rep` type (the internal representation). The rest of the code is self explanatory, and the implementation of the operations is a straight forward implementation of the definition of addition and multiplication for complex numbers.

The Aldor code for the corresponding `Poly` domain can be seen in Figure 5. Again, the `Poly` domain is a sub-type of `Ring`, by augmenting the `Ring` type

```
Ring: Category == with {
    +: (%, %) -> %;
    *: (%, %) -> %;
    0: %;
    <<: (TextWriter, %) -> TextWriter;
}
MI ==> MachineInteger;

Complex: Ring with { complex: (MI, MI) -> %; real: % -> MI; imag: % -> MI }
== add {
    Rep == Record(re: MI, im: MI);
    import from Rep;
    complex(r: MI, i: MI): % == per [r, i];
    real(t: %): MI == rep(t).re;
    imag(t: %): MI == rep(t).im;
    0: % == complex(0, 0);
    (a: %) + (b: %): % == complex(rep(a).re+rep(b).re, rep(a).im+rep(b).im);
    (a: %) * (b: %): % == {
        ra := rep.a; rb := rep.b;
        r := ra.re*rb.re-ra.im*rb.im; i := ra.re*rb.im+ra.im*rb.re;
        complex(r, i);
    }
    (w: TextWriter) << (t: %): TextWriter == w << rep.t.re << "+i*" << rep.t.im
}
```

Fig. 4. Implementation of a simplified complex domain.

with an operation to construct the object. This time the domain is parametric, requiring the type of the coefficients of the polynomial. The internal representation, given by `Rep`, is an `Array(C)`. Here, `Array` is another parametric domain that implements the generic collection similar to arrays in other programming languages. For brevity, we show here only the implementation of the addition between two polynomials, the multiplication and the display functions are similar. The Aldor compiler uses type inference to fill the type information when it is not provided. However, in some cases a disambiguating operator must be used, for example when we specified that `i` gets the value `0` (`0@MI`) from `MachineInteger` rather than the one from `C` or the zero provided by `%`.

An example of usage of the polynomial operations can be seen in Figure 6. The domain constructor creates two polynomials with all coefficients 0, of degree 9999 and then in multiplies them.

For the specialized version, the generic type `C` will be replaced with `Complex`, according to the instantiation given on the first line of Figure 6. The specialized implementation can be seen in Figure 7. One can note that parametric `Poly` domain has become `Poly__Complex`, the internal representation has changed to `TrailingArray`, and `Cross` is used instead of the `Record` for complex numbers. In Aldor, `Record` is a reference type and `Cross` is not, so we perform all data movement locally, without heap allocation. `TrailingArray` is another data type

```
Poly(C: Ring): Ring with { poly: Array(C) -> % } == add {
    Rep == Array(C);
    import from Rep;
    import from C;

    poly(size: MachineInteger): % == {
        res: Array(C) := new(size);
        i:=0@MI;
        while i < size repeat { res.i := 0@C; i := i + 1; }
        per res;
    }
    (a: %) + (b: %): % == {
        c: Array(C) := new(#rep(a));
        i := 0@MI;
        while i < #rep(a) repeat { c.i := rep(a).i + rep(b).i; i := i + 1; }
        per c;
    }
    0: % == poly(1);

    ...
}
```

Fig. 5. Implementation of a simplified generic polynomial type constructor.

```
import from Poly(Complex);
sz == 10000; a := poly(sz); b := poly(sz); d := a * b;
```

Fig. 6. Polynomial multiplication.

supported at the FOAM level. It models a record possibly with a fixed set of leading fields, followed by a set of fields that repeats any desired number of times. This can be used to convert an array of pointers to records to a single structure removing the indirections. The specialized case is presented in Figure 7. It uses both data specialization and code specialization specializations. The code specialization is done by creating a new domain `Poly__Complex`, copying the operations from `Complex` into `Poly__Complex`, and finally inlining the code of complex addition and multiplication into polynomial addition and multiplication.

Data specialization can not be performed in all the cases. If the size of one of the fields is not known it is not possible to inline the data. In some cases, it might be possible to rearrange the field order to keep the variable size structure at the end, but this would still not help once expanded in the outer domain.

The data specialization optimization is still a work in progress, but we have seen that we it is possible to obtain some significant improvements as a result of application of this optimization on top of code specialization optimization that is already implemented.

```
Poly__Complex: Ring with { poly: MI  -> % } == add {
    T == Cross(re: MI, im: MI);
    Rep == TrailingArray(MI, (MI, MI));
    import from Rep;
    import from T;

    poly(size: MI): % == {
        i:MI := 1;
        res: TrailingArray(s:MI,(re:MI,im:MI)) := [size, size, (0,0)];
        while i <= size repeat { res(i, re) := 0; res(i, im) := 0; i := i + 1; }
        per res;
    }
    (a: %) + (b: %): % == {
        local ra: TrailingArray(s:MI,(re:MI,im:MI)) := rep(a);
        local rb: TrailingArray(s:MI,(re:MI,im:MI)) := rep(b);
        res: TrailingArray((s:MI),(re:MI,im:MI)) := [ra.s, ra.s, (0,0)];
        i:MI := 1;

        while i <= ra.s repeat {
            a1 := (ra(i,re), rb(i,re)); b1 := (ra(i,im), rb(i,im));
            aa: Cross(MI, MI) := a1;    bb: Cross(MI, MI) := b1;
            (ar, ai) := aa;             (br, bi) := bb;
            res(i, re) := ar+br;        res(i, im) := ai+bi;
            i := i + 1;
        }
        return per res;
    }
    0: % == poly(1);

    -- Brought from Complex
    local complex(r: MI, i: MI): T == (r, i);
    local a__C(a: T, b: T): T == {
        aa: Cross(MI, MI) := a; bb: Cross(MI, MI) := b;
        (ar, ai) := aa;         (br, bi) := bb;
        complex(ar+br, ai+bi);
    }
    ...
}
```

Fig. 7. Specialized polynomial representation.

6 Results

We modified the Aldor compiler to perform the specialization of domain constructors and exported operations, as described. Table 1 presents the results of testing this optimization. The results vary from one to several times better. Tests one to three are only of depth one, and one can see there is no speedup between the regular Aldor optimizer and our proposed optimization. Tests four to seven use deeply nested types made out of domains that contain simple functions. In

Test	Original Time (s)	Optimized Time (s)	Ratio
Test1	87.13	86.24	1.01
Test2	35.66	35.55	1.00
Test3	35.27	35.27	1.00
Test4	37.71	0.17	∞
Test5	157.78	151.69	1.04
Test6	6.32	0.02	∞
Test7	12.92	1.54	8.39

Table 1. Speedup obtained by automatically specializing the domains.

Test	Original	Optimized	Ratio
Time (s)	119.19	7.98	14.94
Space (MB)	79.6	3.6	22.11

Table 2. Time and run-time memory improvement after hand specialization of polynomial multiplication.

test number four, the difference is big because the regular optimizer does not optimize at all and only our optimization is used. In test five, a simple tower type is used and thus is also optimized by the regular optimizer of the Aldor compiler, but there is still a 4% increase in speedup. Tests six and seven construct the same deeply nested types as in four and five, but they are not fully defined in one place, rather they are constructed in segments. The improvements for tests four and six are too large to measure. These are places were the regular optimizer was unable to optimize. The code specialization optimization does not modify the data representation therefore Table 1 does not mention memory usage.

The tests presented in Table 1 are simple functions that take full advantage of the inline expansion optimization. The next step is to see how this optimization performs on larger functions. An example of the application of this optimization together with the data specialization optimization can be seen in Figure 7.

The results of the specialization applied to the polynomial multiplication problem can be seen in Table 2. For the data representation optimization the creation of objects (mostly temporary objects resulted from arithmetic operations) on the heap is replaced by stack allocated objects and this should produce a decrease in memory usage.

All these tests were performed using Aldor compiler version 1.0.3. The backend C compiler used by the Aldor compiler was gcc 4.1.1. The CPU was a Pentium 4 3.2 GHz with 1MB cache and 2GB RAM. The actual hardware specification is not very important since we are only interested in relative values presented in the ratio columns.

7 Conclusions and Future Work

There are two principal strategies to optimize code: one is from the bottom, as with peep-hole optimization, and the other is from the top, as whole program optimization. When the program is taken as a whole and some properties can be inferred about the code that lead to some very effective optimizations. Program specialization techniques use the second approach to optimization. This second approach can provide significant improvements when it can be applied. The optimization proposed here for Aldor types are of this sort. They could also be applied to Java or C#, which also use a homogeneous approach to implement parametric polymorphism.

Our code and data specialization optimizations could be very well implemented by transforming Aldor source code directly. We chose to use the intermediate representation to take advantage of the existing infrastructure. We note that with nested types the code specialization optimization alone might not bring much improvement. However, with the help of data representation specialization and a nice data structure that allows specialization, as was the case with the polynomial domain, the code can become an order of magnitude faster, even on shallow types.

We have found these results to be sufficiently encouraging that we believe it would be of value to integrate the data representation optimization into the Aldor compiler and to test the compiler using a wider range of real algorithms used in scientific computation.

References

1. S.M. Watt, P.A. Broadbery, S.S. Dooley, P. Iglio, J.M. Steinbach and R.S. Sutor, A First Report on the $A^{\#}$ Compiler, pp. 25-31, Proc. ISSAC 1994, ACM Press.
2. Aldor User Guide, 2000, http://www.aldor.org/
3. S.M. Watt, Aldor, pp. 265-270, in Handbook of Computer Algebra, J. Grabmeier, E. Kaltofen, V. Weispfenning (eds), Springer Verlag 2003.
4. Neil Jones and Carsten Gomard and Peter Sestoft, Partial Evaluation And Automatic Program Generation, Prentice Hall, 1993, ISBN 0-13-020249-5
5. Stephen M. Watt and Peter A. Broadbery and Pietro Iglio and Scott C. Morrison and Jonathan M. Steinbach, FOAM: First Oder Abstract Machine, http://www.aldor.org
6. Laurentiu Dragan, Stephen M. Watt, Parametric Polymorphism Optimization for Deeply Nested Types in Computer Algebra, Maple Summer Workshop, Waterloo, Canada, 2005, ISBN 1-89451-185-9, 243–259.
7. Laurentiu Dragan, Stephen M. Watt, Performance Analysis of Generics in Scientific Computing, Proceedings of Seventh International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, Timisoara, Romania, 2005, ISBN 0-7695-2453-2, 90–100.
8. Jong-Deok Choi, Manis Gupta, Maurricio Serrano, Vugranam C. Shreedhar, Sam Midkiff, Escape Analysis for Java, Proc. OOPSLA 99, ACM Press, 1–19.
9. Alexander A. Stepanov, Appendix D.3 of Technical Report on C++ Performance, ISO/IEC PDTR 18015, 2003