A Mathematical Programming Language Extension for Multilinear Algebra

Felix Friedrich[1], Jürg Gutknecht[1], Oleksii Morozov[2], and Patrick Hunziker[2]

[1] Computer Systems Institute, ETH Zürich, Switzerland
{friedrich,gutknecht}@inf.ethz.ch
[2] University Hospital Basel, Switzerland
{AMorozov,PHunziker}@uhbs.ch

Abstract. Recent results of our ongoing research in the design of a programming language for multilinear algebra are presented. Innovations are the introduction of range-valued indexers, the definition of array-structured object types and the institution of arrays with a dynamic number of dimensions. Together with our conceptual decisions such as the commitment to strict value semantics these concepts help to write highly readable, compact and efficient code. [3]

1 Introduction

In Linear Algebra, elementary objects of computations are expressions of vectors, matrices and tensors. Existing languages do generally not provide a generic support for such terms or require use of unnatural notation, bulky frameworks or intricate libraries. Often, they provide the option of using a high-level but inefficient mechanism or do not support basic safety mechanisms such as range checking. Additionally, we are not aware of a system that intrinsically supports tensors on the level of a programming language.

A simple but nevertheless fundamental example is the implementation of matrix multiplication that should be written in the natural form $C = A \cdot B$ rather than using three nested loops. In most multi-purpose languages the latter is necessary. In mathematical packages, usually the first form is supported but often at the expense of little efficiency when dealing with more generic algorithms. A support of such expressions within a language is highly desirable and from a technical point of view, can be integrated in a straightforward way.

In [2], we have already presented an approach for mathematical programming in linear algebra. We could show that an economic extension of a modern object

[3] **Contributions** F. Friedrich and J. Gutknecht have designed the programming language extension with feedback from P .Hunziker and O. Morozov. F. Friedrich has implemented the necessary compiler- and runtime-support. The compact algorithm for the SVD decomposition (Fig. 2 and Fig. 5) stems from P. Hunziker. O. Morozov has provided the sparse tensor multiplication (Fig. 7) as part of his library on non-uniform spline interpolation of multidimensional signals written in our language.

oriented programming language towards a notation similar to that of Matlab, led to more compact and readable notation while still preserving the high efficiency of a compiled high level language. Moreover, although still being a safe type- and range-checked language, it proved to have very high potential with regards to efficiency for the reason of block-wise operations, the usage of processor vector capabilities, the avoidance of cache misses and the utilization of parallel processors. Above all, fundamental decisions such as the commitment to strict value semantics and pointer-freeness led to a more natural and less error-prone notation.

As an example of our approach, consider the following excerpt from the implementation of a Singular Value Decomposition in Active Oberon (Fig. 1). With the new concepts (such as range-valued indexers and certain operators in the language), the same piece of algorithm can be written much more compactly as displayed in Fig. 2.

```
var u: pointer to array of array of real;
    s,h,f: real; i,j,k,l,m,n: integer;
(* ... *)
for j := l to n do
  s := 0.0;
  for k := i to m  do
    s := s + u[k, i] * u[k, j]
  end;
  f := s / h;
  for k := i to m  do
    u[k, j] := u[k, j] + f * u[k, i]
  end
end;
```

Fig. 1. Small part of SVD in classical notation

```
var u: array [*,*] of real;
    s,h: real; i,j,l,m,n: integer;
(* ... *)
for j := l to n do
  s := u[i..m,i] +*u [i..m,j]; (* scalar product *)
  u[i..m,j] := u[i..m,j] + s/h* u[i..m,i];
end;
```

Fig. 2. New approach: algorithm as in Fig. 1

2 The Language

We decided to implement the language as an extension of (Active) Oberon because Oberon already permits to implement mathematical algorithms in clear and structured form. In principle, however, it should be possible to transfer the ideas to any other imperative programming language in very similar form.

On a first sight, our new arrays are declared and used just like normal arrays in Oberon. The choice of strict value semantics however additionally allows the declaration of dynamic arrays without the unnatural visible notion of pointers (cf. Fig. 3).

2.1 Sub-array Structures

Very often operations are not performed on the complete array but rather on sub-array structures, such as (parts of) columns or rows of a matrix. A first example with operation on rows and columns of a matrix has already been displayed in Fig. 2. The accessibility of substructures of arrays provides the possibility to use optimized vector-oriented algorithms without having to copy parts of the array or to process element-wise. Substructures of arrays are accessed via range-valued expressions like `a..b BY c`. We have provided the star '`*`' for a placeholder of an arbitrary range. Therefore the expressions displayed in Fig. 3 are all valid. Note the automatic type conversion in `Q(v[*])`.

```
var
  M,N: array [*,*] of real;
  v: array [*] of integer;
  w: array [*] of real;

  procedure P(var a: array [*,*] of real);
  procedure Q(const a: array[*] of real);
  procedure R(): array [*] of integer;
begin
  new(M,3,5); NEW(w,10);  (* creation *)
  M := [[1,2,3],[4,5,6]]; (* (re-)allocation if necessary*)
  w[1..3] := M[1,*]; (* assignment *)
  P(M[0..3,0..3]);  Q(v[*]);  v := R(); (* procedure calls *)
```

Fig. 3. Usage of *ranges*

2.2 Operators

A large set of mathematical operators has been defined and implemented, some of them in highly optimized form. The defined operators range from simple unary operators like negation to complex ones like the matrix or tensor product. Examples are given in Figure 4.

```
(* s: scalar; A,B,C: arrays; b: boolean; v: integer vector *)
s := min|max|sum(A); (* array -> scalar *)
A := -B; A := ~ B; A := abs(B); (* array -> array *)
A := short(B); A := long(B); A := entier(B); (* conversion *)
b := B =|<|<=|>|>=|# C; (* array x array -> boolean *)
A := B +|-|*|/|div|mod s; (* array x scalar -> array *)
A := B div|mod|+|-|.*|./ C; (* array x array -> array *)
A := B or|&|.=|.<|.<=|.>|.>=|.# C; (* array x array -> boolean *)
s := B +* C; (* scalar product *) A := B‘; (* transposition *)
A := B * C; (* matrix / vector product *)
A := B ** C; (* tensor product *) A := reshape(B,v); (* reshape operation *)
```

Fig. 4. Some built-in operators; '`|`' denotes an alternative

With the operators and subranges at hand, code can be written closer to mathematical notation and much more compact. As an extreme example observe the compact version of Fig. 1 displayed in Fig. 5.

```
var u: array [*,*] of real; h: real;
(* ... *)
u[i..m,1..n] := u[i..m,1..n]+ u[i..m,i] ** (u[i..m,i] * u [i..m,1..n] / h);
```

Fig. 5. Most compact version of Fig. 1; '**' denotes an outer product

2.3 Custom Array Types

Since not all possible features can be implemented in a built-in array type, we have made provision for the implementation of custom array types. We decided not to arbitrarily extend the functionality of objects to support indexers but merely to provide the concept of *array structured object types* in the language. These object types harmonize with the concept of 'normal' arrays and carry the array structure already in their signature. As an example consider the following layout of an implementation of sparse arrays.

```
type
  Matrix = array [*,*] of real;
  SparseMatrix = array [*,*] of real
    var (* ... data variables etc. *)
    (* allocation and shape *)
    procedure new(i,j: integer);
    procedure len(i,j: integer);
    (* read access *)
    procedure "[]"(i,j: integer): real;
    procedure "[]"(i,j: range): Matrix;
    (* write access *)
    procedure "[]"(i,j: integer; r: real);
    procedure "[]"(i,j: range; const A: Matrix);
    (* ... *)
  end SparseMatrix;
```

Fig. 6. Custom array type definition

2.4 Tensors

The large amount of new articles and independent work in many different *tensor applications* indicates that tensor algebra will develop into a new branch of signal processing. One important branch is the compact representation of multidimensional data in imaging using tensors (cf. [5], [4]).

From the perspective of a programmer, tensors can be viewed as dynamic arrays with an arbitrary dimension. From a mathematical point of view, typical operations, such as the tensor-matrix multiplication, tensor inner- and outer-product and general multiplications along a subset of dimensions, should be supported in the language and form the kernel of the efficient application of algorithms. Both, arrays with dynamic dimensions[4], and a selection of operators and built-in functions are provided by our framework.

Dealing with arbitrary-dimensional arrays means that provision has to be made to dynamically access elements or parts of arrays or the geometric information. We solve this by adding a base-type `range` in the language that can take on integer values or intervals such as `a..b by c` or `*`. Additionally the question mark may be used to denote an arbitrary number of arbitrary ranges.

A non-trivial example of the application of the new constructs in the field of non-uniform spline interpolation of multidimensional signals is displayed in Fig. 7.

[4] We have however not arranged for a discrimination between covariant and contravariant indices of tensors.

It depicts the implementation of the composition of a tensor $\mathbf{G}$ of rank $M \in \mathbb{N}$ by a weighted sum of $N \in \mathbb{N}$ outer tensor-products of $M \in \mathbb{N}$ tensors of rank 1 each (vectors of length $K \in \mathbb{N}$). That is

$$\mathbf{G} = \sum_{i=1}^{N} F_i \cdot \mathbf{B}_{i,1} \otimes \mathbf{B}_{i,2} \otimes \cdots \otimes \mathbf{B}_{i,M}$$

with $\mathbf{B}_{i,j} \in \mathbb{R}^K$ for all $1 \leq i \leq N$, $1 \leq j \leq M$, $F \in \mathbb{R}^N$.
Input of the algorithm is the $N \times M \times k$ ($k \leq K$) array `B` containing the non-zero entries of rank one tensors and the vector `F` denoting the weights. The matrices `L` and `R` for each $1 \leq i \leq N$ and $1 \leq j \leq M$ denote the interval containing non-zero entries of $\mathbf{B}_{i,j}$ and therefore even a sparse tensor multiplication is implemented. The result is written to the tensor `G` with rank (number of dimensions) M. Note how expressions using the question mark can be used to define an iterative application of some function over the number of dimensions.

```
procedure MultiProduct(const F: array [*] of real;
                       const B: array [*,*,*] of real;
                       const L,R: array [*,*] of integer;
                       var G: array [?] of real)
var i,N: integer;

  procedure SubProduct(i: integer; f: real; var g:array [?] of real)
  var j, m, l, r: integer;
  begin
    m := dim(g) -1; l := L[i,m]; r := R[i,m];
    if m > 0 then
      for j := l to r do
        SubProduct(i,f*B[i,m,j-l],g[j,?]);
      end;
    else
      g[l..r] := g[l..r] + f*B[i,0,0..r-l];
    end;
   end SubProduct;

begin
   (* pre: len(F)=len(B,0); dim(G)=len(B,1); len(G,i)=len(B,2) for all i *)
   N := len(B,0) ; (* M = len(B,1); K = len(B,2) *)
   for i := 0 to N-1 do
      SubProduct(i,F[i],G);
   end;
end MultiProduct;
```

Fig. 7. Example of the usage of arbitrarily dimensional arrays

2.5 Implementation and Performance

We have implemented the new arrays following the guidelines of simplicity and safety. For the new dynamic-dimensional array types this meant additional range- and dimensionality-checks. These checks cause an additional runtime cost of about 20 percent for each element-access. However, when using optimized operators on substructures of arrays theses costs vanish and are replaced by performance-boost of a much higher order.

Some of the operators have been highly optimized by using vector capabilities and parallel execution on multi-core machines. The matrix multiplication implementation in Matlab, for example, is very well optimized by using SSE (Streaming Single Instruction Multiple Data Instruction Extension) instructions and an accurately chosen cache aware layout (cf. [1, 3]). Using the same techniques on a single processor machine, we could not do better than that. However, by utilizing the two components of a dual-core processor, we could outperform Matlab by a factor of 1.5. Apparently, the particular Matlab implementation we were comparing with, could not cope with more than one processor for the computation. Standard software can be upgraded to this respect by use of thread-libraries. Although the ATLAS project (Automatically Tuned Linear Algebra Software, cf. [6]), for example, provides a successful example of an automated library optimization, we still argue from our viewpoint of simplicity that the tools for exploiting the mathematical structure of more generic problems should be provided by the language and runtime.

3 Conclusion

We could show that an extension of a language with regards to the handling of typical linear-algebra constructs such as matrices, vectors and tensors can enhance code considerably both with respect to efficiency and readability. Concluding this report we would like to mention that this work – in particular the handling of tensors and custom array types – is subject to ongoing development.

Acknowledgement We thank Luc Bläser for many fruitful discussions and lots of valuable tips.

References

1. D. Aberdeen and J. Baxter. Emmerald: a fast matrix-matrix multiply using Intel's SSE instructions. *Concurrency and Computation: Practice and Experience*, 13(2):103–119, 2001.
2. F. Friedrich and J. Gutknecht. Array-structured object types for mathematical programming. In D. E. Lightfoot and C. A. Szyperski, editors, *JMLC*, volume 4228 of *Lecture Notes in Computer Science*, pages 195–210. Springer, 2006.
3. J. A. Gunnels, G. M. Henry, and R. A. van de Geijn. A family of high-performance matrix multiplication algorithms. In *ICCS '01: Proceedings of the International Conference on Computational Sciences-Part I*, pages 51–60, London, UK, 2001. Springer-Verlag.
4. M. Vasilescu and D. Terzopoulos. Multilinear analysis of image ensembles: Tensorfaces. In *ICPR(2)*, pages 511–514, 2002.
5. H. Wang and N. Ahuja. Compact representation of multidimensional data using tensor rank-one decomposition. In *ICPR '04: Proceedings of the Pattern Recognition, 17th International Conference on (ICPR'04) Volume 1*, pages 44–47, Washington, DC, USA, 2004. IEEE Computer Society.
6. R. C. Whaley, A. Petitet, and J. J. Dongarra. Automated empirical optimizations of software and the ATLAS project. *Parallel Computing*, 27(1–2):3–35, 2001.