Array Operators Using Multiple Dispatch

A design methodology for array implementations in dynamic languages

Jeff Bezanson   Jiahao Chen   Stefan Karpinski   Viral Shah   Alan Edelman

MIT Computer Science and Artificial Intelligence Laboratory

bezanson@mit.edu, jiahao@mit.edu, stefan@karpinski.org, viral@mayin.org, edelman@mit.edu

Abstract

Arrays are such a rich and fundamental data type that they tend to be built into a language, either in the compiler or in a large low-level library. Defining this functionality at the user level instead provides greater flexibility for application domains not envisioned by the language designer. Only a few languages, such as C++ and Haskell, provide the necessary power to define n-dimensional arrays, but these systems rely on compile-time abstraction, sacrificing some flexibility. In contrast, dynamic languages make it straightforward for the user to define any behavior they might want, but at the possible expense of performance.

As part of the Julia language project, we have developed an approach that yields a novel trade-off between flexibility and compile-time analysis. The core abstraction we use is multiple dispatch. We have come to believe that while multiple dispatch has not been especially popular in most kinds of programming, technical computing is its killer application. By expressing key functions such as array indexing using multi-method signatures, a surprising range of behaviors can be obtained, in a way that is both relatively easy to write and amenable to compiler analysis. The compact factoring of concerns provided by these methods makes it easier for user-defined types to behave consistently with types in the standard library.

Keywords   Julia, multiple dispatch, type inference, array indexing, static analysis, dynamic dispatch

1.   Array libraries

> "Unfortunately, it is very difficult for a designer to select in advance all the abstractions which the users of his language might need. If a language is to be used at all, it is likely to be used to solve problems which its designer did not envision, and for which abstractions embedded in the language are not sufficient." - Ref. [22]

n-arrays (arrays of rank n, or simply arrays) are an essential data structure for technical computing, but are challenging to implement efficiently [24, 26, 27]. There is a long history of special-purpose compiler optimizations to make operations over arrays efficient, such as loop fusion for array traversals and common subexpression elimination for indexing operations [3, 24]. Many language implementations therefore choose to build array semantics into compilers.

Only a few of the languages that support n-arrays, however, have sufficient power to express the semantics of n-arrays for general rank n without resorting to hard-coding array behavior into a compiler. Single Assignment C [10] is a notable language with built-in n-array support. Other languages have well-established array libraries, like the C++ libraries `Blitz++` [30] and `Boost.MultiArray` [9] and Haskell's `Repa` (Regular Parallel Arrays) [15, 20, 21]. These libraries leverage the static semantics of their host languages to define n-arrays inductively as the outer product of a 1-array with an $(n-1)$-array [1]. Array libraries typically handle dimensions recursively, one at a time; knowing array ranks at compile-time allows the compiler to infer the amount of storage needed for the shape information, and unroll index computations fully.

1.1   Static tradeoffs

Array libraries built using compile-time abstraction have good performance, but also some limitations. First, language features like C++ templates are not available at run-time, so these libraries do not support n-arrays where n is known only at run-time. Second, code using these features is notoriously difficult to read and write; it is effectively written in a separate sublanguage. Third, the recursive strategy for defining n-arrays naturally favors only certain indexing behaviors. For example, `Repa`'s reductions like `sum` are only defined naturally over the last index [15]; reducing over a different index requires permutations. However, it is worth noting that Haskell's type system encourages elegant factoring of abstractions. While the syntax may be unfamiliar, we feel that `Repa` ought to hold much interest for the technical computing community.

Some applications call for semantics that are not amenable to static analysis. Some may require arrays whose ranks are known only at run-time, e.g. when reading in arrays from disk or from a data stream where the rank is specified as part of the input data. In these programs, the data structures cannot be guaranteed to fit in a constant amount of memory. Others may wish to dynamically dispatch on the rank of an array, a need which a library must anticipate by providing appropriate virtual methods.

1.2   Dynamic language approaches

Applications requiring run-time flexibility are better expressed in dynamic languages such as *Mathematica*, MATLAB, R, and Python/NumPy, where all operations are dynamically dispatched (at least semantically, if not in actual implementation). Such flexibility, however, has traditionally come at the price of slower execution. To improve performance, dynamic languages typically resort to static analysis at some level. One strategy is to implement ar-
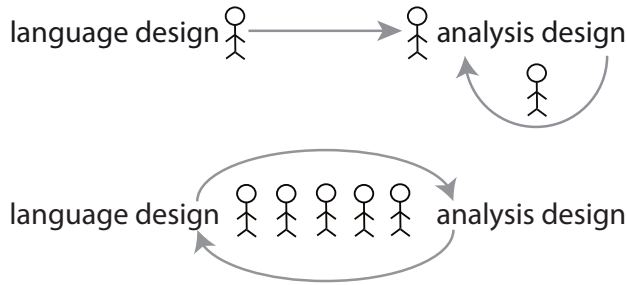
Figure 1. Above: most dynamic languages are designed without consideration for program analysis, leaving it to future compiler writers if the language becomes sufficiently popular. Below: Julia is designed with analysis in mind, with a single community responsible for both language design and performant implementation. This approach is natural for statically typed languages, but dynamic languages need static analysis too.

rays in an external library written in a static language. The Python NumPy package is a prominent example, implementing array operations as well as its own type system and internal abstraction mechanisms within a large C code base [29]. As a result, NumPy `ndarrays` are superficially Python objects, but implementation-wise are disjoint from the rest of the Python object system, since little of Python's native object semantics is used to define their behavior.

Another approach is to implement static analyses *de novo* for dynamic languages. However, the flexibility of these languages' programs limits the extent of analysis in practice. For example, MATLAB's array semantics allow an array to be enlarged automatically whenever a write occurs to an out-of-bounds index, and also for certain operations to automatically promote the element type of an array from real to complex numbers. This poses implementation challenges for static MATLAB compilers like `FALCON`, which have to implement a complete type system with multiple compiler passes and interprocedural flow analyses to check for such drastic changes to arrays [18, 25]. In fact, MATLAB's (and APL's) semantics are so flexible that shape inference on arrays is impossible to compute using ordinary dataflow analysis on bounded lattices [13]. Additionally, type checking is essential to disambiguate MATLAB expressions like `A*B`, which, depending on the dimensions of `A` and `B`, could represent a scaling, inner product, outer product, matrix-matrix multiplication, or matrix-vector multiplication [25]. Similar work has been done for other dynamic languages, as in Hack, a PHP implementation with a full static type system [31].

The conflicting requirements of performance and flexibility pose a dilemma for language designers and implementers. Most current languages choose either to support only programs that are amenable to static analysis for the sake of performance, like C++ and Haskell, or choose to support more general classes of programs, like MATLAB, *Mathematica*, and Python/NumPy. While dynamic languages nominally give up static analysis in the process, many implementations of these languages still resort to static analysis in practice, either by hard-coding array semantics *post hoc* in a compiler, or by implementing arrays in an external library written in a static language.

2. Julia arrays

Julia[2] is dynamically typed and is based on dynamic multiple dispatch. However, the language and its standard library have been designed to take advantage of the possibility of static analysis (Fig-

ure 1), especially dataflow type inference [5, 14]. Such type inference, when combined with multiple dispatch, allows users and library writers to produce a rich array of specialized methods to handle different cases performantly. In this section we describe how this language feature is used to implement indexing for Julia's `Array` data type. The `Array` type is parameterized by an element type and a rank (an integer). For purposes of this paper, its representation can be considered a tuple of a contiguous memory region and a shape (a tuple of integers giving the size of the array in each dimension). This simple representation is already enough to require nontrivial design decisions.

2.1 Array indexing rules

Rules must be defined for how various operators act on array dimensions. Here we will focus on indexing, since selecting parts of arrays has particularly rich behavior with respect to dimensionality. For example, if a single row or column of a matrix is selected, does the result have one or two dimensions? Array implementations prefer to invoke general rules to answer such questions. Such a rule might say "dimensions indexed with scalars are dropped", or "trailing dimensions of size one are dropped", or "the rank of the result is the sum of the ranks of the indexes" (as in APL [7]).

The recursive, one-dimension-at-a-time approach favored in static languages limits which indexing behaviors can be chosen. For example, an indexing expression of a 3-array in C++ might be written as `A[i][j][k]`. Here there are three applications of `operator[]`, each of which will decide whether to drop a dimension based on the type of a single index. The second rule described above, and others like it, cannot be implemented in such a scheme.

Julia's dispatch mechanism permits a novel approach that encompasses more rules, and does not require array rank to be known statically, yet benefits when it is. This solution is still a compromise among the factors outlined in the introduction, but it is a new compromise that we find compelling.

2.2 The need for flexibility

Our goal here is a bit unusual: we are not concerned with which rules might work best, but merely with how they can be specified, so that domain experts can experiment.

In fact, different applications may desire different indexing behaviors. For example, applications employing arrays with units or other semantic meaning associated with each dimension may not want to have the dimensions dropped or rearranged. For example, tomographic imaging applications may want arrays representing stacks of images as the imaging plane moves through a three-dimensional object. The resulting array would have associated space/time dimensions on top of the dimensions indexing into color planes. In other applications, the dimensions are not semantically distinguishable and it may be desirable to drop singleton dimensions. For example, a statistical computation may find it convenient to represent an n-point correlation function in an n-array, and integrate over k points to generate the lower order $(n-k)$-correlation functions; the indistinguishability of the points means that the result is most conveniently expressed with rank $(n-k)$ rather than an n-array with k singleton dimensions.

In practice we may have to reach a consensus on what rules to use, but this should not be forced by technical limitations.

2.3 Multiple dispatch in Julia

Multiple dispatch (also known as generic functions, or multimethods) is an object-oriented paradigm where methods are defined on combinations of data types (classes), instead of encapsulating methods inside classes (Figure 2). Methods are grouped into generic functions. A generic function can be applied to several ar-
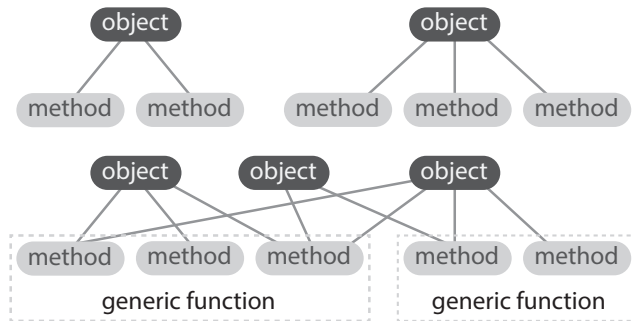
Figure 2. Class-based method dispatch (above) vs. multiple dispatch (below).

guments, and the method with the most specific signature matching the arguments is invoked.

One can invent examples where multiple dispatch is useful in classic object-oriented domains such as GUI programming. For example, a method for drawing a label onto a button might look like this in Julia syntax:

```
function draw(target::Button, obj::Label)
    ...
end
```

In numerical computing, binary operators are ubiquitous and we can easily imagine defining special behavior for some combination of two arguments:

```
+(x::Real, z::Complex) = complex(x+real(z), imag(z))
```

But how much more is there? Would we ever need to define a method on *three* different types at once? Indeed, most language designers and programmers seem to have concluded that multiple dispatch might be nice, but is not essential, and the feature is not often used [23]. Perhaps the few cases that seem to need it can be handled using tricks like Python's `__add__` and `__radd__` methods.

However, in technical computing the need for polymorphic, multi-argument operators goes further. In fact we have found a need for additional dispatch features that are not always found in multi-method implementations. For array semantics, support for variadic methods is perhaps the most important such feature. Combining multiple dispatch and variadic methods seems straightforward, yet permits surprisingly powerful definitions, and entails a surprising amount of complexity. For example, consider a variadic `sum` function that adds up its arguments. We could write the following two methods for it (note that in Julia, `Real` is the abstract supertype of all real number types, and `Integer` is the abstract supertype of all integer types):

```
sum(xs::Integer...)
sum(xs::Real...)
```

The syntax `...` allows an argument slot to match any number of trailing arguments (currently, Julia only allows this at the end of a method signature). In the first case, all arguments are integers and so we can use a naive summation algorithm. In the second case, we know that at least one argument is not an integer (otherwise the first method would be used), so we might want to use some form of compensated summation instead. Notice that these modest method signatures capture a subtle property (at least one argument is non-integral) *declaratively*, without needing to explicitly loop over the arguments to examine their types. The signatures also provide useful type information: at the very least, a compiler could know that all argument values inside the first method are of type `Integer`. Yet

the type annotations are not redundant, but are necessary to specify the desired behavior. There is also no loss of flexibility, since `sum` can be called with any combination of number types, as users of dynamic technical computing languages would expect.

While the author of these definitions does not write a loop to examine argument types, such a loop of course still must take place somewhere inside the dispatch system. Such a complex dispatch system is naturally at risk of performing badly. However, Julia pervasively applies dataflow type inference, so that argument types are often known in advance, in turn allowing method lookup to be done at compile-time. Technically this is just an optimization, but in practice it has a profound impact on how code is written.

2.4 Argument tuple transformations for indexing

Multiple dispatch appears at first to be about operator overloading: defining the behavior of functions on new, user-defined types. But the fact that the compiler "knows" the types of function arguments leads to a surprising, different application: performing elaborate, declarative transformations of argument tuples.

Determining the result shape of an indexing operation is just such a transformation. In Julia's standard library, we have a function `index_shape` that accepts index arguments (which, for present purposes, may be scalars or arrays of any rank), and returns the shape (a tuple of integers) of the result. The length of the shape determines the rank of the result array. Many different behaviors are possible, but currently we use the rule that trailing dimensions indexed with scalars are dropped.[1] For example:

```
A[1:m, 1:n, 2]    # rank 2
A[1:m, 2, 1:n]    # rank 3
A[1:m, 2, 1:n, 1] # rank 3
```

The following two method definitions express this behavior:

```
# drop trailing dimensions indexed with scalars
index_shape(i::Real...) = ()
index_shape(i, I...)    = tuple(length(i),
                               index_shape(I...)...)
```

(The `...` ellipsis syntax within an expression, on the right-hand side of a definition, performs "argument splicing": the elements of a container are spread into multiple arguments to the called function. Formal arguments that lack a `::` type specializer match any value.) The first definition traps and collapses runs of `Real` arguments of any length. The second definition ensures that the first definition only applies to the tail of an argument tuple, by keeping indexes as long as some non-scalar arguments remain.

Since all indexing functions call this function, changing these two lines is sufficient to change how indexing works. For example, another rule one might want is to drop *all* dimensions indexed with scalars:

```
# drop dimensions indexed with scalars
index_shape()           = ()
index_shape(i::Real, I...) = index_shape(I...)
index_shape(i, I...)       = tuple(length(i),
                                  index_shape(I...)...)
```

Or we could imitate APL's behavior, where the rank of the result is the sum of the ranks of the indexes, as follows:

```
# rank summing (APL)
index_shape()       = ()
index_shape(i, I...) = tuple(size(i)...,
                            index_shape(I...)...)
```

---

[1] This rule has been the subject of some debate in the Julia community [11]. Fortunately it is easy to change, as we will see.

Here `size` (as opposed to `length`) gives the shape tuple of an array, so we are just concatenating shapes.

2.5 Exploiting dataflow type inference

Julia's multi-methods were designed with the idea that dataflow type inference would be applied to almost all concrete instances of methods, based on run-time argument types or compile-time estimated argument types. Our definitions exploit the dataflow operation of matching inferred argument types against method signatures, thereby destructuring and recurring through argument tuples at compile-time. As a result, the compiler is able to infer sharp result types for many variadic calls, and optimize away argument splicing that would otherwise be prohibitively expensive. More sophisticated method signatures lead to more sophisticated type deductions.

Tuple types (or product types) are crucial to this analysis. Since the type of each element of a tuple is tracked, it is possible to deduce that the type of `f(x...)`, where x has tuple type `(A,B)`, is equal to the type of `f` applied to arguments of types `A` and `B`. Variadic methods introduce unbounded tuple types, written as `(T...)`. Unbounded tuple types form a lattice of infinite height, since new subtypes can always be constructed in the sequence `(T...)`, `(T,T...)`, `(T,T,T...)`, etc. This adds significant complexity to our lattice operators.

2.6 Similarities to symbolic pattern matching

Julia's multi-methods resemble symbolic pattern matching, such as those in computer algebra systems. Pattern matching systems effectively allow dispatch on the full structure of values, and so are in some sense even more powerful than our generic functions. However, they lack a clear separation between the type and value domains, leading to performance opacity: it is not clear what the system will be able to optimize effectively and what it won't. Such a separation could be addressed by designating some class of patterns as the "types" that the compiler will analyze. However, more traditional type systems could be seen as doing this already, while also gaining data abstraction in the bargain.

2.7 Implications for Julia programmers

In many array languages, a function like `index_shape` would be implemented inside the run-time system (possibly scattered among many functions), and separately embodied in a hand-written transfer function inside the compiler. Our design shows that such arrangements can be replaced by a combination of high-level code and a generic analysis. Similar conclusions on the value of incorporating analyzed library code into a compiler were drawn by the Telescoping Languages project [16]. Yet other languages like Single Assignment C allow great flexibility in user-defined functions but require the built-in shape functions to be implemented with special purpose type functions [10, 28].

From the programmer's perspective, Julia's multi-methods are convenient because they provide run-time and compile-time abstraction in a single mechanism. Julia's "object" system is also its "template" system, without different syntax or reasoning about binding time. Semantically, methods always dispatch on run-time types, so the same definitions are applicable whether types are known statically or not. This makes it possible to use popular dynamic constructs such as `A[I...]` where I is a heterogeneous array of indexes. In such a case the compiler will need to generate a dynamic dispatch, but only the performance of the call site is affected.

One price of this flexibility is that not all such definitions are well-founded: it is possible to write methods that yield tuples of indeterminate length. The compiler must recognize this condition and apply widening operators [5, 6]. In these cases, the deduced types are still correct but imprecise, and in a way that depends on some-

Language	DR	CR	DoS
Gwydion	1.74	18.27	2.14
OpenDylan	2.51	43.84	1.23
CMUCL	2.03	6.34	1.17
SBCL	2.37	26.57	1.11
McCLIM	2.32	15.43	1.17
Vortex	2.33	63.30	1.06
Whirlwind	2.07	31.65	0.71
NiceC	1.36	3.46	0.33
LocStack	1.50	8.92	1.02
Julia	5.86	51.44	1.54
Julia operators	28.13	78.06	2.01

Table 1. Comparison of Julia (1208 functions exported from the `Base` library) to other languages with multiple dispatch. The "Julia operators" row describes 47 functions with special syntax (binary operators, indexing, and concatenation). Data for other systems are from Ref. [23].

what arbitrary choices of widening operators (for example, such a type might look like `(Int...)` or `(Int,Int...)`). Nevertheless, we believe that the flexibility of Julia's multi-methods is of net benefit to programmers.

3. Discussion

Multiple dispatch is used heavily throughout the Julia ecosystem. To quantify this statement, we use the following metrics for evaluating the extent of multiple dispatch [23]:

1. Dispatch ratio (DR): The average number of methods in a generic function.

2. Choice ratio (CR): For each method, the total number of methods over all generic functions it belongs to, averaged over all methods. This is essentially the sum of the squares of the number of methods in each generic function, divided by the total number of methods. The intent of this statistic is to give more weight to functions with a large number of methods.

3. Degree of specialization (DoS): The average number of type-specialized arguments per method.

Table 3 shows the mean of each metric over the entire Julia `Base` library, showing a high degree of multiple dispatch compared with corpora in other languages [23]. Compared to most multiple dispatch systems, Julia functions tend to have a large number of definitions. To see why this might be, it helps to compare results from a biased sample of only operators. These functions are the most obvious candidates for multiple dispatch, and as a result their statistics climb dramatically. Julia is focused on technical computing, and so is likely to have a large proportion of functions with this character.

3.1 Other applications

To give a better sense of how multi-methods are or could be used in our domain, we will briefly describe a few examples.

3.1.1 Array views

In certain instances of array indexing, it is possible to keep the data in place and return just a view (pointer) to the data instead of copying it. This functionality is implemented in a Julia package called `ArrayViews.jl`[19]. A crucial property of an array view is its *contiguous rank*: the number of leading dimensions for which the strides equal the cumulative product of the shape. When a view is

constructed of another view, the type of view constructed depends on the indexes used and the contiguous rank of the argument. In favorable cases, a more efficient `ContiguousView` is returned.

This determination is made by a definition similar to the following:

```
function view(a::DenseArray, I::Subs...)
    shp = vshape(a, I...)
    make_view(a, restrict_crank(contrank(a, I...), shp),
              shp, I...)
end
```

`contrank` essentially counts the number of leading "colons" in the indexing expression. `make_view` selects (via dispatch) what kind of view to return based on the result type of `restrict_crank`, which is set up to return the smaller of its two argument shapes. This is an excellent example of a library that needs to define behaviors actually exceeding the complexity of what is provided in the standard library.

3.1.2 Distributed arrays

Other classes of array indexing rules are needed in distributed array implementations. The Star-P system [4, 12] let users "tag" dimensions as potentially distributed using the notation `*p`, which constructed a special type of object tracked by the system. Indexing leads to questions of whether to take the first instance of `*p` as the distributed dimension, the last instance, or perhaps just the last dimension.

Such distribution rules could be implemented and experimented with readily using an approach similar to that used for `index_shape`.

3.1.3 Unit quantities

A package providing unitful computations (`SIUnits.jl`[8]) makes use of the same kinds of tradeoffs as array semantics. Unitful computations are another case where the relevent metadata (exponents on units) can be known at compile-time in many cases, but not always. The `SIUnits` library is free to express the general case, and have the overhead of tagging and dispatching removed where possible.

The method signatures of operators on unit quantities ensure that they only apply to arguments with the same units. Because of this design, if an operator is applied to two arguments where the units are only statically known for one, the compiler can infer that the other must have the same units for the operation to succeed. Another implication is that mismatched units can be handled with a 1-line fallback definition that simply raises an informative error.

3.2 Performance

In this work, we focus on the performance that arises from eliminating abstraction overhead. Our goal is to convert general definitions (e.g. an indexing function that can handle many kinds of arguments) into the rough equivalent of handwritten C code for any particular case. This is a useful performance target, since programmers often resort to rewriting slow high- level code in C. Further speedups are possible, but entail techniques beyond the current scope. Additionally, we reuse the optimization passes provided by LLVM[17], allowing us to ignore many lower-level performance issues.

Experience so far suggests that we are close to meeting this performance goal [2].

4. Conclusion

Programming languages must compromise between the ability to perform static analyses and allowing maximal flexbility in user programs. Performance-critical language features like arrays benefit greatly from static analyses, and so even dynamic languages that initially lack static analyses eventually want them one way or another.

We speculate that, historically, computer scientists developing multiple dispatch were not thinking about technical computing, and those who cared about technical computing were not interested in the obscurer corners of object-oriented programming. However, we believe that the combination of dataflow type inference, sophisticated method signatures, and the need for high-productivity technical environments is explosive. In this context, multi-methods, while still recognizable as such, can do work that departs significantly from familiar uses of operator overloading. By itself, this mechanism does not address many of the concerns of array programming, such as memory access order and parallelism. However, we feel it provides a useful increment of power to dynamic language users who would like to begin to tackle these and other related problems.

Acknowledgments

References

[1] G. Bavestrelli. A class template for N-dimensional generic resizable arrays. *C/C++ Users Journal*, 18(12):32–43, December 2000. URL http://www.drdobbs.com/a-class-template-for-n-dimensional-gener/184401319.

[2] J. Bezanson, S. Karpinski, V. B. Shah, and A. Edelman. Julia: A fast dynamic language for technical computing. arXiv:1209.5145v1, 2012.

[3] V. A. Busam and D. E. Englund. Optimization of expressions in Fortran. *Communication of the ACM*, 12(12):666–674, 1969. .

[4] R. Choy and A. Edelman. Parallel MATLAB: Doing it right. In *Proceedings of the IEEE*, volume 93, pages 331–341, 2005. .

[5] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '77, pages 238–252, New York, NY, USA, 1977. ACM. .

[6] P. Cousot and R. Cousot. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation. In M. Bruynooghe and M. Wirsing, editors, *Programming Language Implementation and Logic Programming*, volume 631 of *Lecture Notes in Computer Science*, pages 269–295. Springer Berlin / Heidelberg, 1992.

[7] A. D. Falkoff and K. E. Iverson. The design of APL. *SIGAPL APL Quote Quad*, 6:5–14, April 1975. .

[8] K. Fischer. URL https://github.com/loladiro/SIUnits.jl.

[9] R. Garcia and A. Lumsdaine. MultiArray: a C++ library for generic programming with arrays. *Software: Practice and Experience*, 35(2): 159–188, 2005. .

[10] C. Grelck and S.-B. Scholz. SAC: A functional array language for efficient multi-threaded execution. *International Journal of Parallel Programming*, 34(4):383–427, 2006. ISSN 0885-7458. .

[11] T. E. Holy. Drop dimensions indexed with a scalar? URL https://github.com/JuliaLang/julia/issues/5949.

[12] P. Husbands. *Interactive Supercomputing*. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 1999.

[13] P. G. Joisha and P. Banerjee. An algebraic array shape inference system for MATLAB. *ACM Transactions on Programming Languages and Systems*, 28(5):848–907, Sept. 2006. .

[14] M. A. Kaplan and J. D. Ullman. A scheme for the automatic inference of variable types. *Journal of the ACM*, 27(1):128–145, January 1980. .

[15] G. Keller, M. M. Chakravarty, R. Leshchinskiy, S. Peyton Jones, and B. Lippmeier. Regular, shape-polymorphic, parallel arrays in Haskell. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, ICFP '10, pages 261–272, New York, NY, USA, 2010. ACM. .

[16] K. Kennedy, B. Broom, K. Cooper, J. Dongarra, R. Fowler, D. Gannon, L. Johnsson, J. Mellor-Crummey, and L. Torczon. Telescoping languages: A strategy for automatic generation of scientific problem-solving systems from annotated libraries. *Journal of Parallel and Distributed Computing*, 61(12):1803 – 1826, 2001. .

[17] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, pages 75–86, Palo Alto, California, Mar 2004.

[18] X. Li and L. Hendren. Mc2For: a tool for automatically transforming MATLAB to Fortran 95. Technical Report SABLE-TR-2013-4, Sable Research Group, School of Computer Science, McGill University, Montréal, Québec, Canada, 2013. URL `http://www.sable.mcgill.ca/publications/techreports/2013-4/techrep.pdf`.

[19] D. Lin. URL `https://github.com/lindahua/ArrayViews.jl`.

[20] B. Lippmeier and G. Keller. Efficient parallel stencil convolution in Haskell. In *Proceedings of the 4th ACM Symposium on Haskell*, Haskell '11, pages 59–70, New York, NY, USA, 2011. ACM. .

[21] B. Lippmeier, M. M. T. Chakravarty, G. Keller, and S. Peyton Jones. Guiding parallel array fusion with indexed types. In *Haskell '12 Proceedings of the 2012 Haskell Symposium*, pages 25–36, New York, 2012. ACM. .

[22] B. H. Liskov and S. N. Zilles. Programming with abstract data types. In *Proceedings of the ACM SIGPLAN symposium on Very high level languages*, volume 9, pages 50–59, New York, 1974. ACM.

[23] R. Muschevici, A. Potanin, E. Tempero, and J. Noble. Multiple dispatch in practice. In *Proceedings of the 23rd ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications*, OOPSLA '08, pages 563–582, New York, NY, USA, 2008. ACM. .

[24] B. Randell and L. J. Russell. *ALGOL 60 Implementation*, volume 5 of *The Automatic Programming Information Centre Studies in Data Processing*. Academic Press, London, 1964. URL `http://www.softwarepreservation.org/projects/ALGOL/book/Randell_ALGOL_60_Implementation_1964.pdf`.

[25] L. D. Rose and D. Padua. Techniques for the translation of MATLAB programs into Fortran 90. *ACM Transactions on Programming Languages and Systems*, 21:286–323, 1999.

[26] K. Sattley. Allocation of storage for arrays in ALGOL 60. *Communication of the ACM*, 4(1):60–65, 1961. .

[27] K. Sattley and P. Z. Ingerman. The allocation of storage for arrays in ALGOL 60. *ALGOL Bulletin*, Sup 13.1:1–15, 1960.

[28] S.-B. Scholz. Single Assignment C: Efficient support for high-level array operations in a functional setting. *Journal of Functional Programming*, 13(6):1005–1059, 2003. .

[29] S. van der Walt, S. C. Colbert, and G. Varoquaux. The NumPy array: A structure for efficient numerical computation. *Computing in Science & Engineering*, 13(2):22–30, 2011. .

[30] T. L. Veldhuizen. Arrays in Blitz++. In D. Caromel, R. R. Oldehoeft, and M. Tholburn, editors, *Computing in Object-Oriented Parallel Environments*, volume 1505 of *Lecture Notes in Computer Science*, pages 223–230. Springer Berlin Heidelberg, 1998. .

[31] J. Verlaguet and A. Menghrajani. Hack: a new programming language for HHVM. March 2014. URL `http://code.facebook.com/posts/264544830379293`.