Flexible Data Views: Design and Implementation

Leo Osvald Tiark Rompf
Purdue University, USA: {losvald,tiark}@purdue.edu

Abstract

In this paper, we present a library-based framework of data views over chunks of memory segments. Such views not only enable a uniform treatment of references and arrays, but they provide a more general abstraction in the sense that parts of arrays, references, or even views, can be combined into hierarchies to form new logical data structures. To provide efficient implementations in widely used industrial languages such as C++ and Scala, we employ static and dynamic multi-staging techniques, respectively. Through staging and code specialization, the overhead of traversal and tracking of such view hierarchies is mostly eliminated. Thus, our data views can be used as building blocks for creating data structures for which programmers need not pick a specific representation but can rely on code generation and specialization to provide the right implementation that meets asymptotic running time and space guarantees. We apply our approach in case studies in which two-dimensional array views are used to efficiently encode real-world matrices, showing performance on par with specialized data structures such as sparse matrices from popular linear algebra libraries (Armadillo [33] and Eigen [18]), or hand-tuned dense representations. We also show the practicality of specializing data views at run-time on the JVM via Lightweight Modular Staging, a Scala framework for dynamic multi-stage programming, by designing a user-friendly API that hides the underlying compilation through lazy evaluation and a uniform access principle.

CCS Concepts • **Information systems** → **Data access methods**; • **Software and its engineering** → **Data types and structures**; *Patterns*; *Dynamic compilers*; *Source code generation*; *Runtime environments*; Allocation / deallocation strategies; • **Computing methodologies** → Shared memory algorithms

Keywords arrays, specialization, algorithms, memory model

1. Introduction

Programmers often face a choice of how to structure their data, but some choices have long-standing consequences on the code design and, more seriously, performance guarantees. One such dilemma is array versus tuple of same-typed values. An array can be offset using raw pointer arithmetic or *sliced* in order to create subarrays in $\mathcal{O}(1)$ time with no or minimal runtime overhead in some languages, such as C and Go, respectively. A tuple is more syntax-friendly, but conversion to or from an array takes linear time and allocation, forcing a programmer to choose either and be stuck with it.

We consider a more general problem, the design and implementation of *views* on an (ordered) set of data chunks (variables or parts of arrays) without the need for rearranging data in a special way. It should be possible by design that a part of data is seen by multiple views, each providing its own logical layout, and we allow composing views into hierarchies for convenience, therefore our data views must be at least partially (ideally fully) *persistent*.

The simplest type of view we propose is a one-dimensional *array* view, which is basically an ordered collection of chunks of contiguous memory (also called array slices) and/or views themselves. We refer to either of these constituents as view *portions*. A so-called *simple* view is the one in which no portion is another view; Figure 1 shows one such view. By extension, we define N-dimensional (ND) array views as a generalization that supports *logical* layout as of ND arrays (e.g., a matrix if N=2) but with their physical layout hidden. For example, such an abstraction should provide an efficient indexing by coordinates as well as efficient iteration along any of its dimensions. To illustrate that such a problem is not trivial, consider a well-known representation of a sparse matrix in Compressed Sparse Row (CSR) format, which contiguously stores column coordinates of non-zero elements. However, such a representation sacrifices efficiency of column-wise access for a more efficient row-wise access; traversing along a specific column requires some sort of a binary search in each of the rows, and thus requires more than (amortized) constant-time per element. Other formats have their own trade-offs.

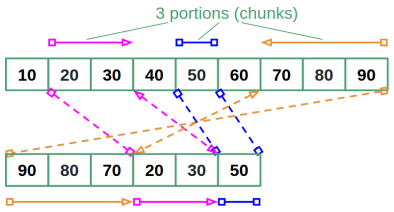

Figure 1. A view (at the bottom) comprising 3 chunks of memory (at the top); the last three elements in the reverse order, the middle element, and the second and third element, respectively.

Instead of settling for a specific representation, we provide a general framework for *specializing* representations of data depending on its *structure* and properties. Some examples are:

- a view that sees every k-th element of an array can be stored as a pair (array a, indexed access function $\lambda i.a[k \cdot i]$);
- tridiagonal matrix as a *composite* view of three 1D array views;
- a view of *immutable* (infinite) series of elements can be represented in $\mathcal{O}(1)$ space using an indexed access function.

When two instances of an ordered data structure are catenated together to form a bigger instance in a *persistent* way (i.e., that both the instances as well as the merger can be accessed), this necessitates multiple levels of nesting in order to avoid decreased performance after many such operations. A simple scenario that results in such a tree-like hierarchy is when a bigger view is repeatedly created out of two or more smaller views. However, having a deep nesting hierarchy hurts performance due to indirection while reading through such composite views. Therefore, we propose using efficient tree-like data structures that we review in Section 7 for nested views, depending on their (statically) declared properties.

As hinted by the examples, using both properties and layout of the data allows for a more efficient access or storage. So, one of

the key ideas in this work is to encode that information into the types. This can be achieved in two ways: *explicitly*, by requiring usage of special types; and *implicitly* via staging, by compiling the code at run-time and evaluating first-stage values, then inspecting the Abstract Syntax Tree and emitting the specialized code in the second stage. In the former case, C++ templates alleviate the burden of pattern matching on types, since the family of closely related types can be represented via a type template (e.g., `Diag<T, BlkHeight, BlkWidth>`) in order to easily refer to their variations with different parameters via function templates that act as meta-functions or in partial specialization (e.g., `template<typename T, size_t...S> Diag<T, Same(S...)>`). The template instantiation allows the compiler to inline certain computation and specialize the code based on the actual template parameters computed at compile-time. In the latter case, the parameters that are only known at run-time must be staged (i.e., evaluated in a later stage), but the rest of the code will be executed and hence inlined or specialized through staging. Compilation at run-time is possible due to the virtualized environment (i.e., Java Virtual Machine). Therefore, the end result is the same, although the latter approach has additional advantages (see Section 5).

Ultimately, using the multi-stage programming framework Lightweight Modular Staging (LMS) [32] in Scala, we support fine-grained specialization of view types at run-time. The compilation overhead is negligible when lots of data is read or written through a view, since we use efficient data structures and view merging algorithms. The whole machinery (staging, code generation and compilation) is hidden from the user by exposing the view framework as a Scala library that relies heavily on lazy evaluation and implicit conversions.

2. Motivating Examples

2.1 Interleaved vs Split Representation

In some numerical libraries that work with complex vectors, such as FFTW [17], Spiral [31] or the C++ STL, APIs expect either of two representations—an array with alternating real and imaginary parts, or the complex and imaginary parts as separate arrays—yet their performance guarantees are sometimes in favor of one or the other. (For example, a null pointer or an array of half the size suffices for the imaginary part in the split representation if the vector is real or conjugate symmetric, respectively.) In those cases, users are forced to do the conversion by copying data, which takes linear time, wastes memory, and requires either provisioning of statically allocated memory for such conversion or paying overhead for a dynamic allocation.

As written in the FFTW documentation, the interleaved format is redundant but still in a widespread use, mostly because it is simpler to use in practice. We introduce an *interleaved* view to neatly provide this convenience without incurring overhead due to conversion between the representation. The index conversion is performed on the fly by division through bit shifting, which should not increase overhead on modern processors that perform both an addition and shifting in one cycle (at least for the cases when array subscripts do not otherwise require bit shifts). In C++, storing such a view as `array<T*, 2>` (i.e., a two-element array of pointers) enables the following implementation of ours for accessing at index i: access the first or the second array (pointer) without branching using subscript $i \,\&\, 1$ (modulo 2), then access the element of type `T` at index $i \gg 1$ (division by 2).

2.2 Excluding a Slice or Combining Arrays

Some algorithms that work with arrays require certain elements to be excluded. Unfortunately, the concept of array slices fails to solve this elegantly because slices can be narrowed but not expanded nor catenated; therefore, one needs to maintain a pair of non-excluded slices instead. To illustrate why this is problematic, consider an algorithm for creating permutations which maintains a list of used elements—eventually a permutation—in array `prefix`, and at each step:

1. picks every unused element stored in array `unused`;
2. solves the problem recursively for modified `prefix` and `unused` with the picked element appended and excluded, respectively.

Observe that a typical implementation would incur $\mathcal{O}(n)$ time overhead to exclude the element by catenating the slices before and after the picked index. Instead, we provide a slice view that is catenable; i.e., two such views (e.g., before and after the excluded element) can be catenated in $\mathcal{O}(1)$ or $\mathcal{O}(\log n)$ time, depending on which guarantees for random access we require, as we are going to explain in Section 4. Additionally, we provide a split operation for our generalization of slice (i.e., a view) into two views, which also runs in *sublinear* asymptotic time. Splitting is especially useful for higher-dimensional views, since widespread representations, e.g., row/column-major (sparse) formats, require linear time.

In both cases, our data views provide the convenience and simultaneously solve the underlying algorithmic challenge of maintaining reasonably efficient, but perhaps irrelevant to the programmer, representation of the accessible data. In cases of catenation and split, the problem boils down to maintaining a balanced or shallow tree (or a forest) of portions, or even provide so-called fingers for more efficient localized access, as well as specialized iterators.

2.3 Sparse Matrices

We show that it matters how views are composed together into hierarchies on the following seemingly toy example[1] of a sparse matrix, which actually comes from a collection of real-world sparse matrices SuiteSparse Matrix Collection [15].

Figure 2 shows a naive breakdown using horizontal then vertical catenation of 2D array views. The sparse matrix comprises: the main diagonal on the left; and the ten parts on the right, each containing a full matrix (whose position vary) and a 3x3 diagonal matrix (at a fixed vertical position). As most elements are on the right, reading through or iterating over such a view involves traversing the view hierarchy of depth 2, and wastes space; i.e., 32 (1+1+10·(1+2)) views are used to represent a sparse 23x63 matrix.

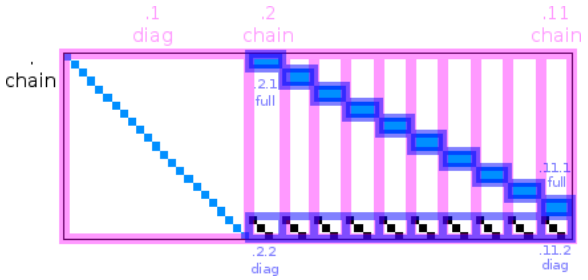

Figure 2. A naive view nesting; each block of the block-diagonal submatrix is catenated with a small diagonal below it, forming vertically nested views (dark blue) that are then horizontally catenated with the main diagonal (magenta) into the outermost view (black).

A more conservative approach is illustrated in Figure 3. Here, the space is saved by observing that full matrices in the top-right corner form a *block-diagonal* matrix; ~50% fewer views are required compared to Figure 2 (15 instead of 32), albeit the small diagonals views are now nested one level deeper (raising the average nesting level from ~1.83 to ~2.05). Moreover, since the blocks are of fixed size (2x4), we are able to optimize away division on

[1] linear programming problem, C.Meszaros test set (p0040)

accesses within such blocks (given a row and/or column) through specialization; for block dimension of size that is a power of two, we do logical shift right (LSR), otherwise we multiply by a magic number that is precomputed statically using C++ templates (or dynamically compiled once on the JVM).

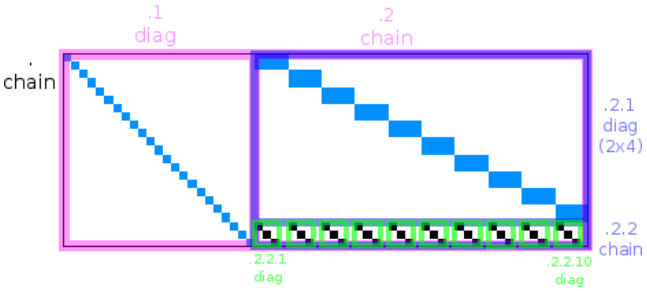

Figure 3. An obvious breakdown into the main diagonal and the rest (purple), which is vertically broken down into a block-diagonal matrix (topmost purple), and a horizontal "chain" of 3x4 matrices (green) with non-zero elements along their main diagonals (black).

In fact, using a more advanced kind of 2D array views we can achieve the same asymptotic complexity of random access and iteration, but decrease the level to 1. The idea is to support a view in which nesting is not necessarily along one dimension (i.e., horizontally or vertically) but may alternate as long as end coordinates of nested views behave as a *monotone* function—this enables binary search in either dimension based on a given row/column to locate the nested views efficiently. Figure 4 illustrates this kind of nesting via a so-called `Mono` view, resulting in only a single level of nesting and 13 views, which is indeed optimal.

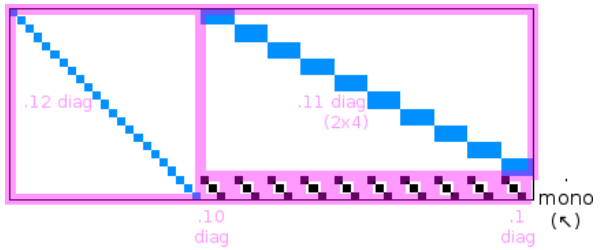

Figure 4. The optimal nesting; all the subviews are catenated at a single level such that their maximal absolute coordinates never increase, in order: the small diagonal views .1 through .10 (the column decreases), the block-diagonal view .11 (the row decreases), and the view spanning the main diagonal (the column decreases).

3. Array Views

Our data views have semantics similar to slices in Go (or the C++ Standard Template Library), except that they can be uniformly used with all built-in data structures such as arrays, plain variables, or even (hash) maps. In addition, we allow combining two or more existing views into a merger view, provided that the corresponding data types are compatible. Lastly, we discriminate between writable and read-only views. As an example of why the last property is desired, consider a view whose data is static, ordered and follows a pattern; in that case, we may use a read-only view that uses $\mathcal{O}(1)$ space and encodes the data using a function. If either of the merged views is read-only, the resulting merger is read-only as well.

Since views can be aliased (i.e., see the shared data), they require some sort of garbage collection. In order to avoid speculating when such resource handling of views is needed, we require that data is only referenced through views, not references nor pointers (i.e., all the variables are views). In that case, it is obvious that the data which can no longer be seen by any view can be deallocated. Conversely, data can be created by expanding a view from a thread; this is a generalization of appending to a slice in the Go programming language (which grows the underlying array). Finally, data can become shared only if another thread creates a view out of the view that uniquely sees it—we refer to such a view as *owner*.

3.1 Higher Dimensions

Our views naturally extend to N dimensions, where we define the following kinds of view via C++ template parameters:

- `NestedArray<T, N>`, a wrapper around `array<T, N>` that provides access by coordinates and iteration along any dimension
- `Sparse<T, N>`, a generalization of a sparse matrix that requires $\mathcal{O}(\log S))$ time for random access, where S is the number of non-default elements (e.g., non-zeros)
- `Diag<T, BlockSizeT, S...>`, a generalization of block-diagonal matrix with $S_1 \times S_2 \times \ldots \times S_N$ blocks
- `Impl<T, N, Access, DimIterFactory>` (usually read-only), which uses $\mathcal{O}(1)$ space by using (stateful) functors (e.g., a closure) for random access and dimension iterator (via a specialized `get<I>` for each dimension I)
- `Chain<T, N, View, Along>`, which catenates views of type `View` into a chain along dimension `Along`; end coordinates for each chained view are required to allow for gaps and/or when dimensionality of nested view is less than $N - 1$
- `Mono<T, N, View>`, which catenates N-dimensional views with monotonically increasing/decreasing end coordinates

All the above family types provide access by coordinates via variadic operator `()`, as well as efficient iteration along any dimension. For `Diag<T, uint8_t, 2, 4>` as an example, random access involves 8-bit arithmetic operations, and dimension iterators maintain a counter which yields a diagonal element when a certain counter value is reached and a default element otherwise.

4. View Run-time

So far, it might have seemed as though views are little more than wrappers around arrays or references. In this section we show that views are, in fact, building blocks for creating self-optimizing data structures. Intuitively, this is possible because data views allow the programmer to specify how they want their data to be accessible and under which asymptotic time and space guarantees but without explicitly choosing a specific representation. Actually, the representation need not even be the same throughout a view's lifetime; e.g., data with the same value can be initially shared but lazily allocated and moved on writes by splitting each affected view into several (as in immutable data structures).

4.1 Representation

As array views are a generalization of slices, they need to store ordered metadata of memory chunks, i.e., triples (source object, begin index, and size or end index). In languages that allow raw memory access via pointers, a pair of virtual addresses unambiguously represents not only an array slice but also a view reference. Otherwise, dummy values for indices (or sizes) can be used but with considerable space overhead. A common base class is a good solution for languages that run in a VM, where virtual dispatch is cheap.

What about nesting? A simple solution is to allow the source object to be a view and use Run-Time Type Information (RTTI) to specially handle cases when a portion is actually a (part of a) view. This works particularly well on the JVM, since `instanceof` checks are very fast, but is neither efficient nor portable in C++; therefore, we use (variadic) template arguments and specialize the cases of array slice/pointers versus views.

4.2 Random Access

Given an index i, the main question is how to efficiently find a portion that sees the i-th element in the imaginary flattened view. If the view is frozen, it might pay off to actually flatten it, and compute the prefix sums of the portion sizes; then the binary search on every random access takes $\mathcal{O}(\log i)$ time, provided that empty views are filtered out during the preprocessing. In the general case, however, a thread may create a view that contains many portions, but the actual amount of accesses through that view is largely dependent on the execution path, which may be much less. Therefore, a conservative choice is to not flatten by default but join the corresponding tree-like nesting hierarchies. Even so, the problem is essentially no different—a binary search along a binary or multi-way search tree may be used, which takes time proportional to the tree depth, especially a self-balancing one such as AVL, Red-Black, or B-tree. Among those tree variants, the AVL tree has the least depth, but in all variants it is straightforward to maintain the subtree size information (which is needed for binary search) without increasing the asymptotic time complexity.

4.3 Iteration

Supporting efficient iteration over a view is tricky because not only portions might be nesting views; they can be views of different kinds! The latter case is particularly problematic because each kind of view has its own iterator type, which means that iteration over a nesting (outer) view requires iteration over nested (inner) views, yet the type of the nested views may change, since the nested view might nest another view, and so on. Therefore, the nested iterators need to be polymorphic. While this does not increase time in the asymptotic sense, it does incur overhead due to virtual dispatch. We forbid empty views, as the iterator's `next` method could otherwise take more than constant time; this way, iteration has the theoretically optimal asymptotic time complexity.

4.4 Split & Exclusion

It is also instrumental to discuss the efficiency of a split operation, which excludes a portion of a view, or (recursively) breaks an existing portion into two portions (i.e., views, respectively). If the AVL trees are used, this operation might not be practically efficient due to a potentially large number of rotations—proportional to the tree height—required to rebalance the AVL tree after deleting a portion (i.e., an element). It has recently been shown by Sen, Tarjan, and Kim [35, 36] that rebalancing need not be performed after the deletion, provided that the such a relaxed AVL tree is periodically rebuilt, without sacrificing logarithmic performance, albeit in terms of insertions in this case.

One of the primary use cases of splitting a view is to decrease or control the aliasing. E.g., if a thread no longer needs part of a view, it might split it at the boundary into two views (and the boundary), and destroy one of them (or the data on the boundary, respectively).

4.5 Catenation (Join)

When two or more array views are catenated (i.e., merged in an order preserving manner), the underlying portion trees undergo a so-called *join* operation, where the indices of the subsequent view operands are increased by the size of the preceding merger. For example, if a view on characters A and B is catenated with a view on character C, the index of C would change from 0 to 2 in the resulting view, but the indices of A and B would remain the same.

View catenation in $\mathcal{O}(1)$ worst case time is possible using persistent deques by Kaplan and Tarjan [24], which also support random access in logarithmic time (as observed by Okasaki [28]). Another data structure that has been shown effective in practice, albeit providing catenation in logarithmic time, is RRB vector [41].

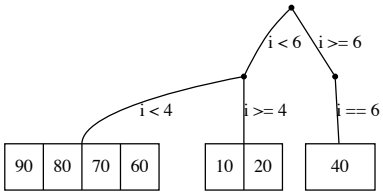

Figure 5. A nested array view with three portions, and decisions for random access through it.

5. Specializing Data Views

As illustrated by motivating examples, naively creating views can result in deep nesting. This is a problem because every random access requires traversal from the root of the corresponding tree down to a leaf, and traversal in general requires polymorphic iteration along the whole tree. In Section 4 we showed a general approach for the most dynamic and unpredictable creation of views, but here we show that we can do much better in many practical scenarios. As an example, consider a view that comprises three array slices of length 4, 3, and 1, respectively, which contains a nested view on the first two chunks as illustrated in Figure 5. (The nesting may have occurred unintentionally, or as a result of catenation for efficiency.) For an efficient access at position i, instead of going through a decision procedure starting from the root towards the leaves—which generally requires $\mathcal{O}(\log n)$ comparisons of i and subtree sizes—we generate a switch table, which is $\mathcal{O}(1)$.

If chunks are indeed statically known, it suffices to use C++ template specialization and metafunctions to create specialized methods for access and traversal of views. In fact, a similar approach is already taken by the Standard Template Library implementors; `vector<bool>` could be considered as a view with a finer granularity—unpacked bits instead of bytes—and `bitset<T, Size>` is specialized into a plain integral type for small sizes.

Otherwise, we use the Lightweight Modular Staging (LMS) framework to specialize the code on the fly. Even though this is expensive, it eventually pays off as we increase the number of accesses to the views, since the specialized code is necessarily more efficient.

Static specialization (using C++ templates) We show static specialization on our block-diagonal array view, which we specialize when block size in every dimension is 1, i.e., it is diagonal:

```
template<class T, typename BlkSizeT, BlkSizeT... S>
class Diag { NestedArray<T, BlkSize, S...>[] bs_;  /* ... */ }
// special case: 1 == S0 == S1 == ... == SN
template<class T, typename BlkSizeT, BlkSizeT S0, BlkSizeT... S,
         typename = enable_if_t<1 == S0 && Same(S0, S...)>>
class Diag<T, BlkSizeT, S0, S...> { T[] bs_;  /* ... */ }
```

In that special case, we use an array to store values along the diagonal, and the rest has some default value (e.g., 0), therefore the access method returns `Same(i...) ? bs_[i] : default_val_`, where `Same` is a variadic function template that checks if all arguments are equal without branching: it statically expands into $(i_0 == i_1)\&(i_1 == i_2)\&\ldots\&(i_{N-1} == i_N)$. In the general case when block sizes are $S_1, S_2, \ldots, S_N$, we store the blocks in a list $bs\_$ of nested arrays that support access by relative coordinates $(i_1, i_2, \ldots, i_N)$. We support random access by absolute coordinates via method `at` implemented as follows,

```
template<size_t Ix0, size_t... Ix, typename I0, typename... I>
T& at0(index_sequence<Ix0, Ix...>, I0&& i0, I&&... i) {
  auto k  =      i0 / get<Ix0>(kScaler);
  return Same(k, (i / get<Ix>(kScaler))...))
    ? bs_[k](i0 - k * get<Ix0>(kScaler),
             (i - k * get<Ix>(kScaler))...) : default_val_; }
static const tuple<DimScaler<S...>> kScaler;
```

```
template<typename... I>
enable_if_t<sizeof...(I)==sizeof...(S), T&> at(I... i) { return
  at0(make_index_sequence_for<I...>{}, forward<I>(i)...);  }
```

which is enabled only if the number of coordinates equals the number of dimensions, and delegates calls to `at0` (and the dummy index sequence $0, 1, \ldots, N$ that only exists at compile time). The `at0` method first computes the index of the block containing the coordinates, k, by dividing block size in any dimension; if quotients are not the same, a default off-diagonal value is returned. It then computes $i \mod S_i$ with a series of logical shifts and additions (instead of multiplications and divisions) in the overloaded operators $*$ and `/` of the helper class `DimScaler`, which is able to specialize this computation because block size S_i is known statically.

We achieve modularity by employing a well-known Curiously Recurring Template Pattern (CRTP). Common functionality (i.e., methods and fields) is statically injected by inheriting one or more helper (base) class templates, each parametrized with an implementation (i.e., `DiagHelper<Derived, ...>`), providing implementation template in terms of `Derived` class. Such *static polymorphism* has no overhead, and helper classes can even access dependent types that may be different in each implementing `Derived` class.

Dynamic specialization (using Scala LMS) A more flexible and user-friendly approach is taken in our implementation of 1D and 2D array views in Scala. The following snippet illustrates the creation of a view on catenation of (reversed) arrays from Figure 5:

```
val a = Array.range(0, 100, 10)  // 0, 10, 20, ..., 90
//  a --(implicit conversion with cache)-> ArrayView
val a9DownTo6And1To2And4V = ArrayView(
    a downTo 6,   a from 1 until 3,   a at 4)
```

Behind the scenes, the `ArrayView` type constructor is a *code generator factory* and its methods (e.g., for random access or iteration) are *lazy* fields that are compiled on first access. For example, reading at index i through the above view is specialized as follows:

```
if (i < 4) a(9-i) else if (i < 6) a(i-3) else a(4)
```

Compared to static specialization, implementation is much simpler because LMS does it automatically for execution paths that do not depend on future-stage values (typed as `Rep[*]`); for example:

```
class Diag[T](bs: Array[ Array[Array[T]] ]) {
  def at(i1: Int, i2: Int): T = atC(i1, i2)
  final lazy val atC = compile2(atS)   // lazily compiled once
  def atS(i1: Rep[Int], i2: Rep[Int]): Rep[T] { // staged
    val (k, k2) = ((i1 / bs(0).size), (i2 / bs(0)(0).size))
    if (k == k2) staticData(bs)(k)(i1 - k * bs(0).size)
                                  (i2 - k * bs(0)(0).size)
    else staticData(defaultVal)                          }
```

where current-staged values such as `bs(0).size` are known during dynamic compilation, so division is optimized away (as in C++).

6. Experimental Results

We have implemented ND array views with static specialization in C++, as well as 1D and 2D array views with dynamic specialization in Scala, as libraries named name *cppviews*[2] and *scalaviews*[3]. During the implementation the main challenge we identified is finding a balance between type refinement and runtime abstractions; the more properties of a view we encode as C++ template parameters or current-staged values (not typed as `Rep[*]` in Scala LMS), the more specialization we need to explicitly deal with. In the former case, apart from the complexity of doing compile-time computation in C++, there is a risk of code explosion. In the later case, not only the JVM may end up compiling too much at run-time, but the space for tuning may grow exponentially and become harder to optimize.

[2] `https://bitbucket.org/losvald/cppviews`

[3] `https://bitbucket.org/losvald/scalaviews`

6.1 Case Study: Strassen Algorithm (Matrix Multiplication)

The Strassen algorithm is an efficient divide-and-conquer algorithm for matrix multiplication in time $\mathcal{O}(N^{\log_2 7+o(1)}) \approx \mathcal{O}(N^{2.8074})$, which is faster than the naive $\mathcal{O}(N^3)$ algorithm. The asymptotic improvement in time is achieved by partitioning either square matrix (to be multiplied) into 4 equally sized block matrices—here is where our views come into play—and thus reducing the number of multiplications from 8 to 7. Our baseline is a fast C/C++ implementation by Cochran [12] in which partitioning is done in $\mathcal{O}(1)$ time by adjusting the access strides for the submatrices, but this makes the implementation verbose as both strides and offsets of block matrices need to be explicitly recalculated and carried around. Instead, we represent submatrices with views and split them (in $\mathcal{O}(1)$ time) at each step in the recursion. Table 1 presents the results, from which we can see that our convenient and conceptually simple approach has only 20% slowdown for sufficiently big matrices.

N	256	512	1024	2048	4096
strides & offsets	0.012	0.116	0.585	4.015	30.303
splittable views	0.017	0.136	0.704	4.827	36.660
relative slowdown	44%	16%	20%	20%	21%

Table 1. Running time in seconds of two implementations of the Strassen algorithm, a hand-optimized one that explicitly calculates strides as well as offsets (to avoid copying) and ours in which dense views are simply split, for multiplying two NxN matrices.

6.2 Case Study: Real-world Sparse Matrices

We have visually examined a huge collection of real-world sparse matrices from SuiteSparse [15], and observed that many can be represented using the same kind of views and with nestings of similar depths. We selected a matrix of sufficient size (typically hundreds of thousands of elements) as a representative of each such equivalence class, as well as some of the atypical ones in order to stress test our methods. Details of the matrices can be found through the online search tool[4] by entering their unique names.

We were able to represent each of our sample matrices using 2D array views defined in Section 3.1 with only a few levels of nesting (depicted as magenta, dark blue, green, red, respectively), after allowing ourselves to: waste a small fraction of space by over-approximating certain submatrices as dense by using full views, which is shown in Figure 6; or potentially give up some performance by using sparse views instead of fully exploiting a structure of a submatrix with complicated patterns, as illustrated in Figure 7.

To evaluate performance of reading sparse matrices through our views, we first wrote a GUI program (with an interface similar to the previous figures), which generates a JSON file that describes the user-created view hierarchy without the actual non-zero elements; i.e., which views cover which parts of the matrix and how they are nested into the top-level view. Then, we have a C++ code generator that outputs a header file in which views have many properties statically encoded using C++ template parameters, as shown in Listing 1, so that further compilation for the benchmark of a particular view specializes the code. For each 3rd-party library that we compare performance against, we wrote a template-specialized sparse matrix view facade, `SmvFacade<ThirdPartySparseMatrix>`, which allows for easy uniform and static treatment. The overhead of the facade layer is normally optimized away by the C++ compiler, since our classes use *static polymorphism* and their methods simply delegate parameters to the APIs of the underlying libraries. Figure 8 shows the part of our evaluation pipeline that produces `*.hpp` header files that declare an uninstantiated class template of a view-like object (each inheriting the corresponding facade), and Figure 9 shows the next

[4] `http://yifanhu.net/GALLERY/GRAPHS/search.html`

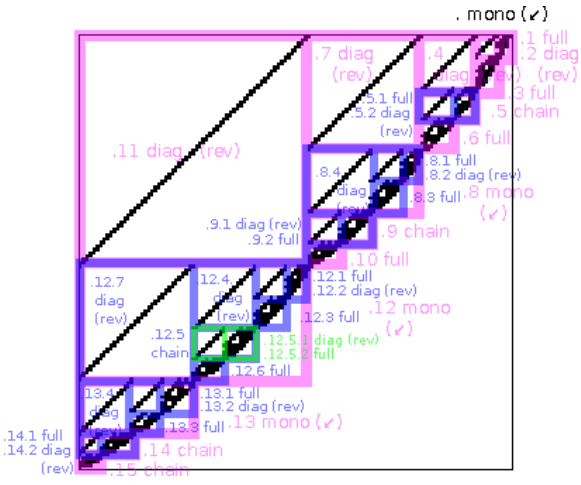

Figure 6. Simplical complexes from Homology from Volkmar Welker (n3c6-b7). The parts around the antidiagonal are represented via 16 full views (`NestedArrays`), each of approximate size as the rightmost green rectangle, although these small submatrices look similar to the whole matrix (i.e., have a fractal pattern).

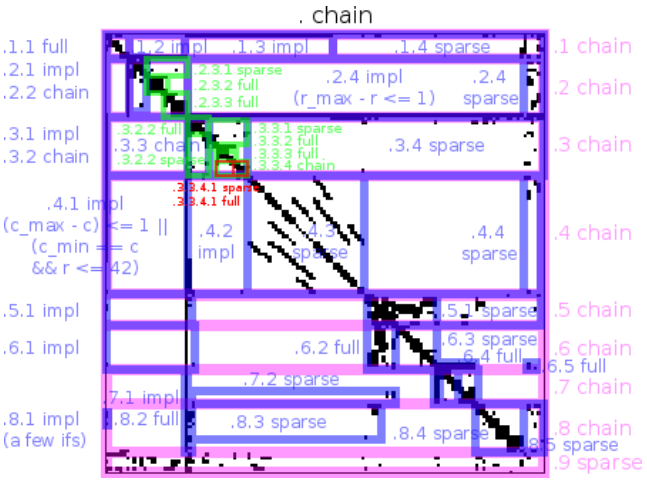

Figure 7. A circuit simulation problem (rajat01). The central region with diagonal-like submatrices—not even block-diagonal due to gaps (not visible)—is underapproximated by using a sparse view. This avoids the need of a nearly-(block-)diagonal kind of view.

stage in which the code is specialized (through template instantiation and specialization) based on a statically known view hierarchy (or properties of sparse matrices in case of 3rd-party libraries).

Using our pipeline, we performed a series of microbenchmarks, random reading of zero and non-zero values, and iterating over non-zero values in a fixed order (consistent with iteration over the corresponding indices). We measured average times on 3–5 runs of these benchmarks on two matrices—containing 133 and 255004 non-zero elements (and 23 rows and 32 columns, and 60008 rows/-columns, respectively)—such that a large number of candidate access coordinates are precomputed (typically 10^5–10^6 pairs), which are repeatedly shuffled and read in round-robin fashion sufficiently many times (10^6–10^9) in each run, so that the times are around a second. Table 2 and Table 3 show normalized results for these two matrices in millions of IO operations per second (IOPS). In a sufficiently large matrix, our random access of non-zero values is 658% and 14% faster than the one of sparse matrices in Armadillo and Eigen libraries, and 77% faster than the C++ hash table, while

```
#include "facade.hpp"
struct Figure3
#define SM_BASE_TYPE Chain<ArrayView<int, 2>, 1>
    : public SM_BASE_TYPE, public SmvFacade<Figure3> {
  Figure3() : SM_BASE_TYPE(       // CRTP ^^ for static
#undef SM_BASE_TYPE               // injection of methods
    ChainTag<1>(), PolyVector<ArrayView<int, 2>>()
    .Append([] {  // MAIN DIAGONAL
      Diag<int, uint, 1, 1> v(ZeroPtr<int>(), 23, 23);
      for (uint i = 0; i < 23; ++i) v(i, i) = 1;
      return v; }())
    .Append(  // VERTICALLY CHAINED RIGHT PART (DARK BLUE)
      ChainTag<0>(), PolyVector<ArrayView<int, 2>>()
      .Append([] {// 2x4 BLOCK-DIAGONAL
        Diag<int, uint, 2, 4> v(ZeroPtr<int>(), 20, 40);
        return v; }())
      .Append(    // HORIZONTALLY CHAINED DIAGONALS (GREEN)
        ChainTag<1>(), PolyVector<ArrayView<int, 2>>()
        .Append([] {
          Diag<int, uint, 1, 1> v(ZeroPtr<int>(), 3, 3);
          for (uint i = 0; i < 3; ++i) v(i, i) = -1669;
          return v;  }())
        // ... 9 more Appends with different values ^^ ...
        , ZeroPtr<int>(), ChainOffsetVector<2>({
            {0, 0}, /* ... 8 more pairs ... */ {0, 36}   })
        , 3, 40)
      , ZeroPtr<int>(), ChainOffsetVector<2>({{0, 0}, {20, 0}})
      , 23, 40)
    , ZeroPtr<int>(), ChainOffsetVector<2>({{0, 0}, {0, 23}})
    , 23, 63) {  // INITIALIZATION OF VALUES
    static int data[] = { -1, -1, -1, -1,  +1, +1, +1, +1,
                          // ... for 8 more blocks ...
                          -1, -1, -1, -1,  +1, +1, +1, +1, };
    static uint rows[] = { 0,  0,  0,  0,   1,  1,  1,  1,
                          // ... for 8 more blocks ...
                          18, 18, 18, 18,  19, 19, 19, 19, };
    static uint cols[] = {23, 24, 25, 26,  23, 24, 25, 26,
                          // ... for 8 more blocks ...
                          59, 60, 61, 62,  59, 60, 61, 62, };
    for (size_t i = 0; i < 80; ++i)
      (*this)(rows[i], cols[i]) = data[i];                } }
```

Listing 1: Generated C++ header code (except include guards) for the view in Figure 3. `Diag` views have their block sizes, 2x4 and 1x1, as template parameters, which enables shifts by a constant instead of divisions upon random access. Similarly, chaining dimensions are statically encoded via `ChainTag` for efficient iteration.

	Random Read		Iteration
	0s	non-0s	non-0s
`arma::SpMat`	110.536	104.984	450.352
`Eigen::SparseMatrix`	71.180	39.701	1745.864
cppviews	44.389	34.972	34.620
	44.155	37.026	33.155
`std::map`	30.941	27.067	196.826
`std::unordered_map`	39.718	56.875	–

Table 2. Performance of random reads and iteration in millions of IOPS (higher is better), for the small sparse matrix p0040. Two implementations of the view hierarchy as in Figure 3 were benchmarked, in which the sizes of chained diagonals as well as gaps in between are both statically known or unknown, respectively. Creating either view using our GUI tool took 35–60 seconds on average.

the random access to zero values is still acceptable, 59% and 93% slower than in Armadillo and Eigen. Iteration over non-zero values is also several times slower, which is understandable since we do not statically eliminate nesting in our C++ implementation.

7. Related Work

The View Template Library [44] is the most closely related work we are aware of; it implements views in C++ as container adaptors which provide access to different representations of data that are generated on the fly. Such a concept view is a generalization of a *smart iterator* [6], which can filter data while iterating over a data structure (i.e., a container), as opposed to providing a transformed access over it. A *live* data view [7] has also been studied in the context of parallel and mobile environments as a programming abstraction of a time window over streaming data.

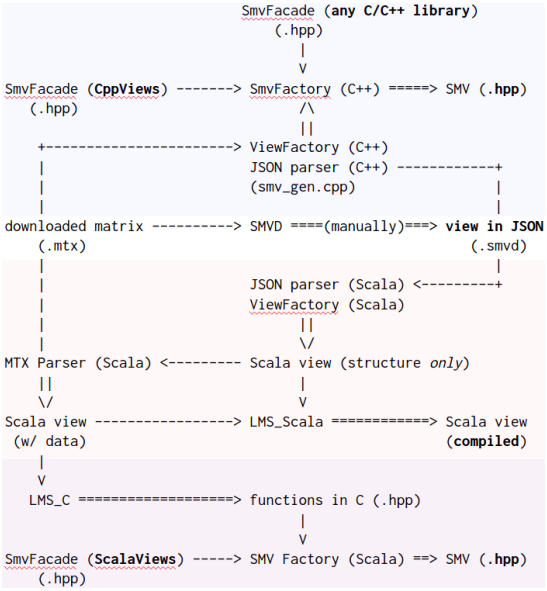

Figure 8. The pipeline for generating a specialized view of a sparse matrices as a C++ header file out of its JSON representation, dynamically from Scala (lower part), or statically from C++ code (upper part) using cppviews or a third-party library that provides matrix-like data structure for which a facade should be written.

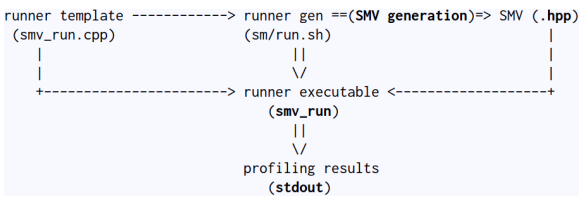

Figure 9. Evaluation pipeline for running experiments using the generated C++ header file (see Figure 8).

	Random Read		Iteration
	0s	non-0s	non-0s
`arma::SpMat`	51.259	2.757	172.665
`Eigen::SparseMatrix`	57.814	15.872	251.166
cppviews	29.997	18.149	45.075
`std::map`	2.023	1.497	55.155
`std::unordered_map`	8.937	10.252	–

Table 3. Performance of random reads and iteration in millions of IOPS (higher is better) on the large sparse matrix a5esindl, which we represent with `Diags` nested up to 3 levels deep via `Chain` views. The view creation using our GUI tool took 3–5 minutes on average.

Persistent Data Structures A general framework for turning ephemeral pointer-based data structures into persistent ones was provided by Driscol et al. [16] and improved by Brodal [8].

Arrays The concept of an array for contiguous storage has been introduced by Konrad Zuse [5], and Fortran was the first language that implemented it. *Discontiguous* arrays divided into indexed chunks have been proposed by several researchers [20, 4, 11, 38], and have been extensively studied in the scope of virtual machines, where fragmentation caused by large arrays results in unpredictable space-and-time performance during garbage collection. To reduce fragmentation, Siebart [38] groups such chunks into a tree, but this requires an expensive tree traversal on every access. Bacon et al. [4], Pizlo at al. [29], and Sartor et al. [34] use a single level of indirection to fixed size *arraylets*. Sartor et al. further reduce the indirection overhead by a constant factor via their first-N optimization, and use other optimization techniques such as zero compression, lazy allocation, and arraylet copy-on-write [34].

Trees Kuszmaul [26] provides a technique for merging balanced binary trees in $\mathcal{O}(1)$ amortized time. Red-black trees were invented by Guibas and Sedgewick [19], and remain one of the few balanced search trees in which rebalance after every operation requires $\mathcal{O}(1)$ rotations in the worst case (including the deletion). AVL trees [1] have remained one of the most rigidly balanced trees ever since their introduction in 1962, and require at most two rotations per insertion. That a deletion in an AVL tree can cause $\Theta(\log n)$ rotations, even in the amortized case, has been proved by Amani et al. [3]. Sen, Tarjan and Kim [35, 36] recently described a relaxation of AVL and other balanced binary search trees in which deletions do not rebalance the tree at all, yet worst-case access time remains logarithmic in the number of insertions, provided that it is periodically rebuilt. For cases when access is localized, faster trees exist; Hinze and Paterson invented 2-3 trees known as Finger trees [21], which are purely functional and designed with simplicity of implementation in mind. Finally, some trees were invented to perform better on *non-uniform* access patterns; their amortized time to access an item v is in $\mathcal{O}(1 + \log \frac{m}{c(v)})$, which matches the theoretical optimum [25] as a function of access frequencies $c(v)$. The earliest such is the splay tree by Sleator and Tarjan [40]. In a recent work with Tarjan, Yehuda et al. [2] devised the CB tree—a practical concurrent alternative that achieves the same asymptotic optimality—in which the number of rotations is *subconstant* amortized if the majority of operations are lookups and/or updates (not insertions).

Lists Skip lists [30] are a simpler and significantly faster alternative to traditional self-balancing search trees, but with the same asymptotic expected time bounds (i.e., $\mathcal{O}(\log n)$) proved by randomized analysis. They support catenation and splitting. A purely functional *random-access* list that supports $\mathcal{O}(\min\{i, \log n\})$ time lookup or update at index i, and stack operations in $\mathcal{O}(1)$ time, was presented by Okasaki [28]. If external immutability suffices, there are simpler fully persistent random access deques that rely on memoization and lazy evaluation to achieve amortized $\mathcal{O}(1)$ deque operations including catenation, in addition to access in $\mathcal{O}(\log i)$ amortized time, as shown by Brodal et al. [23]. The RRB vector [41] is a random-access deque that supports appending/deleting at either end in amortized $\mathcal{O}(1)$ time, catenation and lookup/update at index in $\mathcal{O}(\log n)$, but exploits spatio-temporal locality.

Metaprogramming Siek and Taha [39] formalize semantics of C++ templates, which provide a Turing-complete sub-language within C++ through specialization. Cole and Parker [13] develop a method for *dynamic* compilation of C++ templates that delays code generation for instantiated templates until they are actually used at run-time. Multi-staged programming (MSP) was pioneered by Taha [43], mostly through MetaML [42] that allows code generation at run-time. Czarnecki et al. [14] show how to implement Domain Specific Languages (DSLs) using MSP: dynamically in MetaOCaml [9], but also statically in Template Haskell [37] and C++ via template metaprogramming. Lightweight Modular Staging (LMS) [32] is a Scala library for MSP that relies solely on types to distinguish the computational stages, unlike previous approaches—MetaML [42] and MetaOCaml [9]—which rely on quasiquotes. Scala LMS is inspired by Carette et al. [10] and Hofer et al. [22], can generate the code at run-time, and allows for *deeply embedded* DSL implementation through Scala Virtualized [27].

8. Conclusion

We design and implement data views that are more general than existing data structures, supporting efficient operations such as split/-catenation in N dimensions. They allow not only finer-grained resource management, alias control and sharing; they shift the burden

of picking the optimal representation from a programmer to the compiler. In C++, we compared our view performance and found it superior to optimized implementations of general-purpose data structures such as hash and sorted maps, and not more than a few times slower than ad-hoc (domain-specific) representations implemented in state-of-the-art linear algebra libraries. For efficiency, we use static specialization, static polymorphism and other compile-time metaprogramming facilities. Finally, we show feasibility of dynamic specialization through on-line compilation in Scala LMS, a multi-stage-programming framework, on our prototype library that hides the complexity of specialization from the user.

Acknowledgments

Special thanks to Jan Vitek for insightful suggestions and guidance in the early stages of this project. Thanks to David Gleich for pointing us to SuiteSparse. This research was supported by NSF through awards 1553471 and 1564207.

References

[1] G. M. Adelson-Velskii and E. M. Landis. An algorithm for the organization of information. In *Soviet Mathematics Doklady 3*, pages 1259–1262. 1962.

[2] Y. Afek, H. Kaplan, B. Korenfeld, A. Morrison, and R. E. Tarjan. The CB tree: a practical concurrent self-adjusting search tree. *Distributed Computing*, 27(6):393–417, 2014.

[3] M. Amani, K. A. Lai, and R. E. Tarjan. Amortized rotation cost in AVL trees. *Information Processing Letters*, 116(5):327–330, 2016.

[4] D. F. Bacon, P. Cheng, and V. T. Rajan. A Real-time Garbage Collector with Low Overhead and Consistent Utilization. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, volume 48, pages 58–71, New York, NY, USA, jul 2013. ACM.

[5] F. L. Bauer and H. Wössner. The PlankalkÜL of Konrad Zuse: A Forerunner of Today's Programming Languages. *Commun. ACM*, 15(7):678–685, jul 1972.

[6] T. Becker. Smart Iterators and STL. *C/C++ Users J.*, 16(9):39–45, sep 1998.

[7] J. Black, P. Castro, A. Misra, and J. White. Live data views: programming pervasive applications that use "timely" and "dynamic" data. *MDM '05: Proceedings of the 6th international conference on Mobile data management*, pages 294–298, 2005.

[8] G. S. Brodal. *Worst Case Efficient Data Structures*. PhD thesis, Department of Computer Science, Aarhus University, Denmark, 1997.

[9] C. Calcagno, W. Taha, L. Huang, and X. Leroy. *Implementing Multi-stage Languages Using ASTs, Gensym, and Reflection*, pages 57–76. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003.

[10] J. Carette, O. Kiselyov, and C.-C. Shan. Finally Tagless, Partially Evaluated. In Z. Shao, editor, *Programming Languages and Systems: 5th Asian Symposium, APLAS 2007, Singapore, November 29-December 1, 2007. Proceedings*, volume 19, pages 222–238. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

[11] P. Cheng and G. E. Blelloch. A Parallel, Real-time Garbage Collector. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, PLDI '01, pages 125–136, New York, NY, USA, 2001. ACM.

[12] W. Cochran. Program that tests the performance of the Volker Strassen algorithm for matrix multiplication. `https://ezekiel.encs.vancouver.wsu.edu/~cs330/lectures/linear_algebra/mm/`, 2011.

[13] M. J. Cole and S. G. Parker. Dynamic compilation of C++ template code. *Scientific Programming*, 11(4):321–327, 2003.

[14] K. Czarnecki, J. T. O'Donnell, J. Striegnitz, and W. Taha. DSL Implementation in MetaOCaml, Template Haskell, and C++. In C. Lengauer, D. Batory, C. Consel, and M. Odersky, editors, *Domain-Specific Program Generation: International Seminar, Dagstuhl Castle, Germany, March 23-28, 2003. Revised Papers*, pages 51–72. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[15] T. A. Davis and Y. Hu. The University of Florida Sparse Matrix Collection. *ACM Trans. Math. Softw.*, 38(1):1:1–1:25, 2001.

[16] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan. Making data structures persistent. *Journal of Computer and System Sciences*, 38(1):86–124, 1989.

[17] M. Frigo. A Fast Fourier Transform Compiler. In *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, PLDI '99, pages 169–180, New York, NY, USA, 1999. ACM.

[18] G. Guennebaud, B. Jacob, et al. Eigen v3. http://eigen.tuxfamily.org, 2010.

[19] L. J. Guibas and R. Sedgewick. A dichromatic framework for balanced trees. *19th Annual Symposium on Foundations of Computer Science (sfcs 1978)*, pages 8–21, 1978.

[20] B. Hess, T. R. Gross, and M. Püschel. Automatic locality-friendly interface extension of numerical functions. In *Proceedings of the 2014 International Conference on Generative Programming: Concepts and Experiences*, GPCE '14, pages 83–92, New York, NY, USA, 2014. ACM.

[21] R. Hinze and R. Paterson. Finger trees: a simple general-purpose data structure. *Journal of Functional Programming*, 16:197–217, 2006.

[22] C. Hofer, K. Ostermann, T. Rendel, and A. Moors. Polymorphic Embedding of DSLs. In *Proceedings of the 7th International Conference on Generative Programming and Component Engineering*, GPCE '08, pages 137–148, New York, NY, USA, 2008. ACM.

[23] H. Kaplan, C. Okasaki, and R. E. Tarjan. Simple confluently persistent catenable lists. *Algorithm Theory - Swat'98*, 1432(3):119–130, 1998.

[24] H. Kaplan and R. E. Tarjan. Persistent Lists with Catenation via Recursive Slow-down. In *Proceedings of the Twenty-seventh Annual ACM Symposium on Theory of Computing*, STOC '95, pages 93–102, New York, NY, USA, 1995. ACM.

[25] D. E. Knuth. *The Art of Computer Programming, Volume 3: (2Nd Ed.) Sorting and Searching*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1998.

[26] L. Kuszmaul. Efficient Merge and Insert Operations for Binary Heaps and Trees. *STAR*, 38, 2000.

[27] A. Moors, T. Rompf, P. Haller, and M. Odersky. Scala-virtualized. *Proceedings of the ACM SIGPLAN 2012 Workshop on Partial Evaluation and Program Manipulation*, pages 117–120, 2012.

[28] C. Okasaki. Purely functional random-access lists. *Proceedings of the 7th international conference on Functional programming languages and computer architecture*, 33(Sep):86–95, 1995.

[29] F. Pizlo, L. Ziarek, P. Maj, A. L. Hosking, E. Blanton, and J. Vitek. Schism: Fragmentation-tolerant Real-time Garbage Collection. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, pages 146–159, New York, NY, USA, 2010. ACM.

[30] W. Pugh. Skip Lists: A Probabilistic Alternative to Balanced Trees. *Commun. ACM*, 33(6):668–676, jun 1990.

[31] M. Püschel, F. Franchetti, and Y. Voronenko. Spiral. In D. Padua, editor, *Encyclopedia of Parallel Computing*. Springer, 2011.

[32] T. Rompf and M. Odersky. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs. *Commun. ACM*, 55(6):121–130, 2012.

[33] C. Sanderson and R. Curtin. Armadillo: a template-based C++ library for linear algebra. *The Journal of Open Source Software*, 1:26, 2016.

[34] J. B. Sartor, S. M. Blackburn, D. Frampton, M. Hirzel, and K. S. McKinley. Z-rays: Divide Arrays and Conquer Speed and Flexibility. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, pages 471–482, New York, NY, USA, 2010. ACM.

[35] S. Sen and R. E. Tarjan. Deletion Without Rebalancing in Balanced Binary Trees. In *Proceedings of the 21st Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '10, pages 1490–1499, Philadelphia, PA, USA, 2010. SIAM.

[36] S. Sen, R. E. Tarjan, and D. H. K. Kim. Deletion Without Rebalancing in Binary Search Trees. *ACM Trans. Algorithms*, 12(4):57:1–57:31, sep 2016.

[37] T. Sheard and S. L. Peyton Jones. Template meta-programming for Haskell. *SIGPLAN Notices*, 37(12):60–75, 2002.

[38] F. Siebert. Eliminating External Fragmentation in a Non-moving Garbage Collector for Java. In *Proceedings of the 2000 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, CASES '00, pages 9–17, New York, NY, USA, 2000. ACM.

[39] J. Siek and W. Taha. *A Semantic Analysis of C++ Templates*, pages 304–327. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.

[40] D. D. Sleator and R. E. Tarjan. Self-adjusting Binary Search Trees. *J. ACM*, 32(3):652–686, jul 1985.

[41] N. Stucki, T. Rompf, V. Ureche, and P. Bagwell. RRB Vector : A Practical General Purpose Immutable Sequence. 32:342–354, 2015.

[42] W. Taha and T. Sheard. MetaML and multi-stage programming with explicit annotations. *Theor. Comput. Sci.*, 248(1-2):211–242, 2000.

[43] W. M. Taha. *Multistage programming: its theory and applications*. PhD thesis, 1999.

[44] M. Weiser and G. Powell. The View Template Library. In *In First Workshop on C++ Template Programming*, 2000.