C♭: A Modular Approach to Efficient and Tunable Collections

STEPHAN BRANDAUER, Uppsala University, Sweden
ELIAS CASTEGREN, Uppsala University, Sweden
TOBIAS WRIGSTAD, Uppsala University, Sweden

Collections are commonly implemented as libraries by data structure experts, and are relied on heavily by application developers. The expert's task is to implement a wide range of collections, and the application developer's task is to pick an appropriate collection for each usage scenario. The design space for collections is huge, as data structures in practice implement not only their semantics, but also several performance-related concerns like memory layout, synchronisation, and (im)mutability.

This paper presents C♭, pronounced "C-flat", a novel way to implement collections that lets experts implement the semantics of a collection data structure, in a way that is decoupled from its data representation. This simplifies collection implementation, and allows a collection's performance to be tuned, for example, moving from a dense to a sparse representation, without changing its abstract specification.

We describe C♭, both abstractly and in terms of a specific prototype implementation in Java. We use our prototype implementation to show that C♭ is expressive enough to implement common collections, that the code is straightforward, and that the performance of C♭ collections is close to Java's standard collections for most operations, and much higher for some.

ACM Reference Format:
Stephan Brandauer, Elias Castegren, and Tobias Wrigstad. 2018. C♭: A Modular Approach to Efficient and Tunable Collections. 1, 1 (July 2018), 20 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

This paper presents a novel approach to implementing collection data structures. Different collections have different strengths and weaknesses with respect to performance under different usage scenarios. For example, although equivalent from the abstract data-type point of view (because they support the same operations), array lists offer constant-time lookup, but suffer from linear complexity for insertion and removal at the front or in the middle of a list, whereas linked lists offer constant-time prepend and append, but linear time lookup. Optimising a collection to improve performance therefore commonly amounts to replacing one concrete implementation of an abstract data type with another, *e.g.,* using an `ArrayList` instead of a `LinkedList` in Java.

The novel approach taken in this paper is driven by a desire to separate functional and non-functional concerns (we focus on run time and memory required) for collections. This simplifies their respective implementation, and allows collections to be optimised without touching the "business logic" satisfying the functional requirements. We propose an approach for constructing collection data structures where each collection (a *front-end*) is implemented against an abstract representation of memory (*a cursor*) that can be freely combined with different *back-ends*, which implement the actual physical storage of data. When front-ends and back-ends are combined freely, non-functional

Authors' addresses: Stephan Brandauer, Uppsala University, Sweden, stephan.brandauer@it.uu.se; Elias Castegren, Uppsala University, Sweden, elias.castegren@it.uu.se; Tobias Wrigstad, Uppsala University, Sweden, tobias.wrigstad@it.uu.se.

XXXX-XXXX/2018/7-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

properties (performance and memory required) can be picked according to application needs, while functional properties remain invariant. As a consequence of this modular design, development of a new back-end allows the creation of as many new collections as there are front-ends, and bug fixes or performance improvements in one back-end or front-end is enjoyed by all combinations that rely on it.

This paper makes the following contributions.

(1) We propose a novel approach to implementing collection data structures, which separates the functional requirements from the non-functional requirements, and allows these to vary independent of each other. The same back-end can serve different front-ends, *e.g.,* lists, trees, matrices, etc. The same front-end can use different back-ends, *e.g.,* array-based, dense, sparse, etc. We describe this approach in § 2.
(2) We define a simple language for expressing a collection's logical representation, from which a compiler can generate a cursor (§ 2.3) that translates operations on the logical representation (in the front-end, § 2.2) to the physical representation (on the back-end, § 2.4).
(3) We present a prototype implementation of our ideas in Java, including several front-ends and back-ends, without needing to extend the Java language, or requiring extra compilation steps other than using Java's annotation processing. This implementation is described in § 3.
(4) We evaluate our ideas *qualitatively* in § 4.1 through the implementation of several front-ends (that implement existing Java abstract collection interfaces like `List` and `HashMap`) and several back-ends. In particular, we show the low complexity of code in C♭ front-ends.
(5) We evaluate our ideas *quantitatively* in § 4.2 and § 4.3 through a performance comparison between combination of front-ends and back-ends with concrete counterparts in JDK 1.8. In particular, we demonstrate that the performance of C♭ collections can be close to the Java collections, which have been developed over many years. We also show that in the performance can be *tuned* to greatly improve performance of specific use cases, like sparse matrix multiplication.

2 C♭ IN A NUTSHELL

In traditional data structure implementation, a collection's operations and physical representation – and hence performance considerations – are conflated. C♭ untangles these aspects by letting the programmer express the behaviour in terms of operations on a logical representation of memory, using a cursor, which are translated into operations on the physical representation.

For a library implementer, C♭ shifts the problem of providing a versatile set of collections to providing a versatile set of front-ends that target specific functional requirements and a set of back-ends that target specific non-functional requirements, which can then be combined for a multitude of different scenarios. Thus, a library implementer might provide back-ends that trade memory for performance, or the other way around. The library implementer could rely on traditional collections to implement back-ends, but since back-ends have a very limited interface, implementing back-ends from scratch is significantly simpler than implementing collections with rich interfaces.

For an application programmer, using C♭ libraries is indistinguishable from traditional libraries, except when the existing set of libraries is not enough. If additional functionality is sought, she can develop new front-ends that provide additional user-facing operations while relying on the existing back-ends for efficient storage. If existing back-ends do not meet performance requirements, she can implement a new specialised back-end and use it with the existing front-ends, knowing that the latter are operationally unaffected (but hopefully perform better).

Additionally, the clear separation between front-end and back-end lowers the barrier for performance optimisations such as changing the back-end of a collection at run-time based on profiling data. A modern language built with C♭ might easily support this as part of a JIT'ing infrastructure.

2.1 Design Overview

This section describes the design of C♭. A C♭ collection consists of three components:

(1) a *front-end* that implements some abstract interface, such as a list, tree or matrix;
(2) a *back-end* that implements how the memory that front-ends require will be physically stored in memory—for instance, as an array, a tree, or a sparse matrix; and
(3) a *cursor* that connects the front-end with the back-end without tying the implementation of either to the other. This is key to freely combining any front-end with any back-end.

C♭ collections are restricted to collections whose spines can be described as trees. This guarantees the existance of a unique path to every element in the collection. The interface between the front-end and back-end is expressed in terms of such paths. Back-ends present data as trees to the front-end, but may store the data in a different form. Changing how elements are stored in a collection without touching its front-end is a core contribution of this work.

For concreteness, consider the two collections in Fig. 1. To the left is a list represented as a 1-ary tree where each node holds an element. To the right, a matrix is represented as a 2-ary tree whose right branch holds a row and whose left branch holds the remaining columns of the matrix.

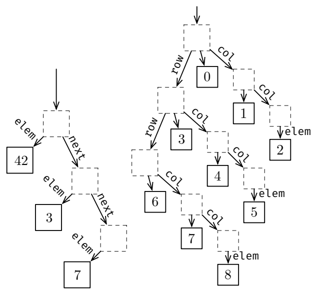


Fig. 1. A list and a matrix represented as trees. The representation of these is an implementation detail with a wide range of choices. C♭ front-ends are implemented in terms of a *logical representation* which may be mapped to different *physical representations*, *c.f.*, Fig. 4.

Because of the imposed tree shape constraint, the path from the root of the outermost tree to any element in a collection is unique. If, in the list, we name each recursive edge in the tree *next* and the edge leading to the payload *elem*, then, to get to the value 3, the path is *next.elem*. In the matrix, if we call each recursive edge leading to a new row *row*, each edge traversing further into a row *col*, and each edge leading to a number *elem*, the path *row.col.col.elem* takes us to the value 5.

Logically, a C♭ back-end is a map from paths to storage locations which may or may not hold a value. To access the element at row 1, column 2 in the matrix in Fig. 1, the front-end would ask for the datum at *row.col.col.elem*, which abstractly describes how to move through the matrix to obtain the requested element.

Because each path is unique and enumerable, an implementation can represent each path by an integer[1]. The prototype implementation in this paper takes this approach. This compact representation allows efficient implementation of the back-ends, and maps directly to indices of certain back-ends. For example, the path *row.col.col.elem* is (regardless of the back-end's implementation) mapped to 5, which would be the expected index if the back-end was implemented as one big array. How a path maps to an index in the general case is specified in § 2.3.

Having given a high-level idea of what C♭ is about, we are now ready to go into details on the three ingredients – front-ends, cursors, and back-ends.

2.2 Front-Ends: A Collection's Operations

Front-ends express their operations in terms of the abstract paths introduced in the previous section. Importantly, by using abstract paths, front-ends do *not specify how the internal representation will be actually layed out in memory; layout is the job of the back-end.* What abstract paths are legal

[1]Other options for expressing the interface between front-ends and back-ends exist. § 6 describes cases where using integers has disadvantages, and how to make back-ends that overcome these disadvantages.

Table 1. (Left) C♭ syntax. $S, S_1, S', \ldots$ denote step names, *e.g.,* `next`, `left`. (Right) Examples.

C ::=	*(C♭ expression.)*
$\mid \star(S) \rightarrow S'$	Repeat S unboundedly, then S'.
$\mid \star N(S) \rightarrow S'$	Repeat S upto N times, then S'.
$\mid \; \Vert(A_1, \ldots, A_n)$	Pick one alternative in $\{A_1, \ldots, A_n\}$.

C♭ Expression	*Meaning*
$\star$(next) $\rightarrow$ elem	List
$\star$7(next) $\rightarrow$ elem	List, length $\leq$ 7.
$\star$(‖(left, right)) $\rightarrow$ elem	Binary tree.
$\star$($\star$7(child) $\rightarrow$ pick) $\rightarrow$ elem	7-ary tree.
$\star$3(row) $\rightarrow$ $\star$3(col) $\rightarrow$ elem	3x3 matrix.

for a front-end is controlled by a *C♭ expression*, which can be thought of as a small grammar for generating valid paths, abstract locations where data may be stored.

Tab. 1 shows the syntax of C♭ expressions, along with a few examples and their interpretations. C♭ expressions come in three forms: unbounded repetitions $\star(\ldots) \longrightarrow \ldots$, bounded repetitions $\star N(\ldots) \longrightarrow \ldots$, and disjunctions ‖(. . . , . . .). C♭ expressions serve two important purposes. First, they let a programmer define and capture the abstract representation of the collection that she will rely on to implement the collection's behaviour. Second, they allow straightforward code generation of cursors, *i.e.,* the glue code that lets front-ends and back-ends communicate.

A partial implementation of a list collection, `Sequence`, is found in Fig. 2. As it relies heavily on cursors, it makes sense to first look at it briefly and then return to it after § 2.3.

2.3 Cursors: The Interface between Front-Ends and Back-Ends

Consider a binary tree data structure, described by the expression $\star$(‖(`left`, `right`)) $\longrightarrow$ `elem`. A back-end must implement how to map all paths (*e.g.* `left.left.elem`, or `right.elem`) matching the expression to data, by providing a way to store and retrieve data at a path. As a back-end can provide storage for any front-end, it cannot assume knowledge of the front-end's C♭ expression. To break the seeming dependency that back-ends have on the C♭ expression they are hosting data for, we use a concept that we call cursors. A cursor can be thought of as a feature-rich iterator object synthesised to work with a particular abstract representation.

A front-end developer manipulates a cursor to construct paths: she instantiates a cursor object, which is initialised to refer to the root of the data structure, and then calls "step methods" on the cursor instance to iterate through the data structure. Step methods advance a cursor according to the C♭ expression by following a "field" (one can think of calling a step method as appending to a path). For example, a cursor for the expression $\star$(`next`) $\longrightarrow$ `elem` has a step method `next()` that advances the cursor by one step each time it is called. When the cursor reaches a position of interest, the programmer can ask the back-end to store or to retrieve data stored at that location. Fig. 2 shows examples of this in the `add()` method, where the `set()` operation is used directly on the back-end, without committing to a particular representation (*c.f.,* § 2.4). Sometimes, multiple or temporary cursors are useful, *e.g.,* when copying elements between data structures.

An implementation of C♭ is free to choose how to map paths to back-end locations—our implementation follows the convention that disjunctions map each of their inner subexpressions to consecutive integer ranges: ‖(`FALSE`,`TRUE`) would map `FALSE` to 0, `TRUE` to 1; ‖($\star$7(`NEXT`) $\longrightarrow$ `GET`, `NONE`) would map the 7 choices in the left subexpression to the range $[0, 6]$, and `NONE` to 7. Repetitions (using the $\star$ operator) use an encoding similar to an array-based implementation of a binary heap: no traversals of the repeating subexpression map to 0, one traversal maps to the next N locations, two traversals to the next N^2 locations, etc. Using integers to represent paths is useful because it makes back-ends have an interface that is easily implemented and optimised.

The cursor class also has convenience methods that move the cursor back one step. For instance, if `next()` advances the cursor by one step, `next_back()` will move it one step back. For trees, `left_back()` will move the cursor back to the parent node.

```
@Cflat("*(next)->elem") // Used to generate the SequenceCursor
class Sequence<T> implements java.util.List {
  private Backend<T> data; // The back–end (†)
  SequenceCursor tail = new SequenceCursor(); // The cursor abstracting the back–end
  boolean add(T obj) {
    data.set(this.tail.elem(), obj); // Store elem at the current cursor location
    tail.next(); // Position cursor after new element
    return true; // part of java.util.List interface
  }
  boolean contains(Object obj) {
    // return whether any element equals obj
    return data.nonNullIndices().findAny((T)obj) != -1;
  }
}
```

Fig. 2. Programming against a logical representation: Sequence (most important methods).

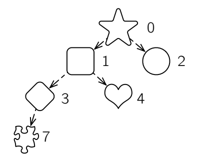
Even though a collection is abstractly described as a tree, movement or iteration over its elements is not constrained by the tree-shape. For example, column-wise iteration is possible in a matrix, just as one would expect. The performance of such operations depends on the choice of back-end. As an example of a step method, consider the `left()` method of a binary tree cursor (from the expression $*(\|(\texttt{left}, \texttt{right})) \longrightarrow \texttt{elem}$). Fig. 3 shows the code generated automatically in our C♭ prototype. (Returning `this` permits call chaining.)

```
public BinTreeCursor left() {
  this.pos = this.pos * 2 + 1;
  return this;
}
```

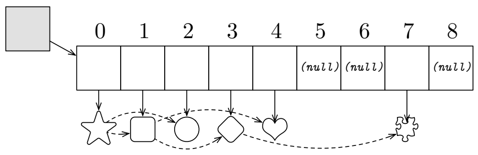
Fig. 3. A step method that advances a binary tree cursor to its left child.

For cases where a C♭ expression contains unbounded subexpressions, integer indexing is no longer sufficient. Consider the expression $*(\texttt{row}) \longrightarrow *(\texttt{col}) \longrightarrow \texttt{elem}$. As each row has a potentially infinite number of values, there is no specific integer that would denote a node in any row except the first. That is to say, that the paths $\texttt{col} \ldots \texttt{col}$ all would have a defined index; $\texttt{row.col} \ldots \texttt{col}$ however would not. An implementation of C♭ has several options to handle expressions with unbounded subexpressions[2]. Our implementation currently picks the simplest one and refuses to accept expressions with unbounded subexpressions. Instead, the front-end developer can use two simpler cursors (equal in shape to the sequence's cursor) together with a back-end that contains back-ends, where each inner back-end represents a row. This corresponds to the view of the matrix as an array containing arrays (the type in Java would be `T[][]` for some element type `T`). While this might seem overly limiting, and while investigating other options has merit, nested back-ends (*c.f.*, § 2.4.2) unlock useful possibilities for back-end implementers.

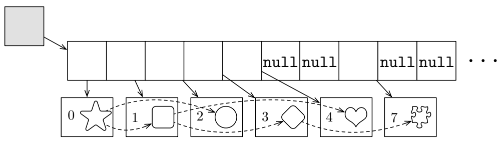
[2] *Option 1*: Projecting the expression at runtime. An implementation can require a maximum value for the number of columns *at runtime*. This means that the implementation views the expression as really defining a family of C♭ expressions, with a limited number of columns: $\forall C \in \mathbb{N}, *(\texttt{row}) \rightarrow *C(\texttt{col}) \rightarrow \texttt{elem}$, where C is picked appropriately by the data structure's front end at runtime. The integer that the path `row.col.col.elem` would map to with C columns picked at runtime would be $1 * C + 2$ – skipping all the C indices reserved for the first row and picking the `col.col` child of the second row. *Option 2: Using multi-dimensional coordinates.* An implementation can also represent a cursor in such a data structure as a tuple of integers that denote the consecutive children that were picked at each node; the expression `row.col.col.elem` would be represented as the multi-dimensional index [1, 2] (meaning $1 \times \texttt{row}$, $2 \times \texttt{col}$). Back-ends could use integer arrays as position arguments, or the tuple could be converted to a single integer using a reversible $\mathbb{N}^2 \leftrightarrow \mathbb{N}$ mapping. Arrays would have extra allocation costs and mappings would not work equally well for all data structures, which is why we do not explore this option here.



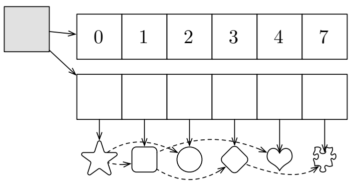
(a) *An example binary tree.* Data are annotated with their index.



(b) *The tree in (a) using an array back-end.* Dashed lines depict the logical connections in the original example.



(c) *The tree in (a) using a hash map back-end.* This is more space-efficient for sparse data sets than the array back-end.



(d) *The tree in (a) using a sorted-array back-end.* This is even more space efficient than a map, has excellent iteration performance, but costly inserts.

Fig. 4. A collection's front-end can be connected to different representations, for performance.

2.4 Back-Ends: Controlling the Storage of Elements

Back-ends are data structures that map paths to locations storing values. Depending on how the `data` field is instantiated on Line † in Fig. 2, the sequence will perform *storage* differently, but its *behaviour* is invariant. Looking at Fig. 1, one might think that a linked data structure is not a good representation for a matrix, as accessing a value in the matrix would have the complexity $O(\text{rows} + \text{columns})$. Luckily, back-ends are at liberty to implement the physical storage of data in any way they want, regardless of the front-end's logical representation (the dashed-boxes in Fig. 1). For instance, the matrix could be stored as a single array, or an array of arrays. It could also be represented as a sparse matrix that only stores the non-empty cells. If none of the existing back-ends work well for an algorithm, a user can implement her own back-end – and reuse it with any front-end.

Back-ends must map locations to values in a consistent fashion: storing a value at the location corresponding to a path and subsequently accessing the same location should always give the same result. Below, we discuss different ways in which one might consider implementing back-ends.

2.4.1 Flat Back-Ends. Flat back-ends are back-ends that store application data, as opposed to nested back-ends that store *back-ends* (§ 2.4.2). This section gives a brief overview of important back-ends, but we introduce more in the evaluation.

Dense/Sparse Back-Ends. The memory requirement of a *dense* back-end grows with the range of the input. For example, an array-based back-end (see Fig. 4b) maps paths to consecutive storage locations, allowing paths to be used directly as array indices. This gives fast random access, fast iteration over values in a back-end, and effective utilisation of cache-locality and hardware prefetching.

A downside of dense back-ends generally, and array-based back-ends specifically, is that even unused array cells use memory. Consider the binary tree in Fig. 4b: the tree is mapped onto the array, but values that are not present in the tree still require memory to store **`null`** references. This characteristic can make an array-based back-end a bad fit for sparse collections where only few values are defined with large gaps in between. An example of such a sparse data structure is the hash-map implementation in § 4.1.2.

In contrast, the memory requirement of a *sparse* back-end depends on the number of non-null values stored in it, not their location. For example, a hash map-based back-end (that stores the values as values of a hash map that uses the locations as keys[3], see Fig. 4c), sorted array backend (that stores the values in an array in index order and the values' locations in a parallel integer array, see Fig. 4d), or trie-based backend (not drawn, but it divides the key into 4 consecutive bytes and uses those bytes as indices into a lazily initialised 256-ary tree for fast random access, but relatively high upfront memory requirements) keep track of storage locations in use, making them consume memory proportional only to the number of values stored.

Mutable/Immutable Back-Ends. Immutable back-ends never change their data; instead, modifying operations return updated versions of the back-end. Immutable back-ends support efficient implementation of copying the entire back-end: the identity function. For example, if data is frequently copied across threads to avoid data races, using an immutable back-end may be advantageous. In C♭, a data structure developer can easily implement collections that can be switched from mutable to immutable by their user by simply employing an immutable back-end. There are several different strategies for implementing immutable versions of data structures [Okasaki 1996]. The immutable data structure that we use to implement an immutable back-end in our prototype is a *Compressed Hash-Array Mapped Prefix Tree* (CHAMP), [Steindorfer and Vinju 2015]. This data structure is a map, and is therefore well-suited for sparse data.

2.4.2 Nested Back-Ends. A nested back-end has elements which are also back-ends. They may give more freedom to implement useful data layouts than using the back-ends outlined above.

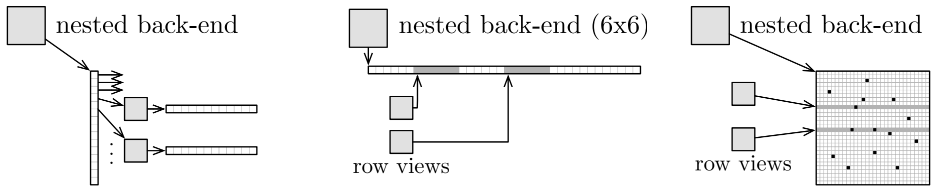
To illustrate, consider a $R \times C$ matrix: it could be implemented by using a flat back-end as storage, and – assuming row major order (WLOG) – the front-end would map the first row's data to the indices $[0..R)$, the second row's data to the indices $[R..2 \cdot R)$, etc. But at this point, the front-end would, once again, hard-code the data layout and the front-end would now forever be forced to represent its data in row-major order. This problem can be avoided by nesting back-ends. Using nested back-ends, a user can implement a matrix front-end that looks as if it was a row-major matrix, yet still supply it with a back-end that will in fact implement a physical column-major layout (or more complicated layouts, like sparse matrices).

All the flat back-ends (like the ones presented up to this point) can be used as nested back-ends, where there is one outer back-end containing back-ends representing rows of data. Examples would be an array-back-end containing array-back-ends for dense data (Fig. 5a), or a map-based back-end containing array-back-ends as an optimisation for use cases where most rows are empty. Even though flat back-ends can be combined to form nested back-ends, there can be nested back-ends that do *not represent their internal rows as separate, disjoint, back-ends.*

As a trivial example, consider a $n \times m$ matrix that is mapped onto a single large array of size $n \times m$ (Fig. 5b). This layout could not be produced by using a flat back-end that contains other flat back-ends, as this would lead to an array that contains references to disjoint arrays. Such a back-end could, however, be still implemented by implementing a nested back-end directly, that for each row access returns a *view of the inner large array* that implements all operations of a back-end. Each view would store the row index r it represents, and accessing a location c of a row would then access the location $r \cdot m + c$ in the large array.

One non-trivial example are certain sparse matrix data structures (we implement the so-called *compressed sparse row* format of a sparse matrix [Buluç et al. 2009]) that can serve as a nested

[3]In Java such a hash map with integer keys can be faster than a hash map with a generic key. If a hash map in Java has a generic type parameter for the key type, this implies that the keys are represented as *references to objects* of the `Integer` class, rather than compact primitive `int` values.



(a) A nested back-end, constructed from flat back-ends (array back-ends, in this case).
(b) A hypothetical back-end that organises its rows in consecutive ranges of an array.
(c) A nested back-end that uses a sparse matrix representation.

Fig. 5. A nested back-end may return row-views that can, in concert, operate on a large, shared data structure.

back-end using views. Accessing any value in this row view will, in fact, access the sparse matrix representation. Fig. 5c depicts this idea: the nested back-end contains a reference to a sparse matrix. Rows are, again, just views onto parts of the data structure.

Nested back-ends have the advantage that layout optimisations can be "global": empty columns in our sparse matrix based nested storage consume no memory at all, and empty rows consume very little. They can also be helpful to avoid pointer indirections, as Fig. 5 shows. Yet, they still are compatible with code that uses flat storages. This is important: the sequence data structure we implement in § 4.1.1 can, for instance, be used to turn a row view of a matrix into a list, permitting a front-end developer to operate on a nested back-end's rows using algorithms like `java.util.Collections.sort`, that are defined with the list interface. For example, this lets a front-end developer sort rows of any data structure by composing code, rather than writing her own sort implementation.

This concludes our high-level introduction to C♭: a separation of collection data structures into two components glued together by a third. Programmers use C♭ expressions to capture the logical representation of data in a collection, and program against this logical structure in the front-end. The same expressions are used to generate glue code to connect the front-end to a back-end. Back-ends can be implemented freely, as long as they support consistent mapping from paths to values. For many logical strucutures, it is possible to represent paths as integers, which facilitates efficient implementation. Next, we describe our prototype implementation that we use to evaluate our ideas in practice.

3 PROTOTYPE IMPLEMENTATION

We implement C♭ on top of Java, together with a selection of back-ends. Our implementation of C♭ is using Java's annotation framework. Specifically, we do *not* rely on a modified Java compiler, rather just use tools that are available in a standard Java environment. We embed C♭ expressions in Java code through Java annotations. Our C♭ prototype consists of two major parts:

(1) An annotation processor that processes C♭ expressions at compile time and turns them into "Cursor classes", Java classes that let a user express paths in the C♭ language, see § 3.1.
(2) A library of back-end implementations. These implementations implement a common interface that provides ways to set and access data at a cursor-defined location, as well as iterate over data, see § 3.3.

3.1 C♭ Expressions and The Java Annotation Processor

A Java annotation processor is a user-defined class that picks up annotations as code is being compiled. The annotation processor can then generate code that will be available during the compilation phase of a Java project. For C♭, a user can annotate a Java class with an annotation to describe the class'

Table 2. 1) Memory-inefficient. 2) Explained in Sec. 2.4.2. 3) Cannot use nested back-end here.

Back-End	Front-End Compatibility			Performance Characteristics
	Seq.	Map	Matrix 2)	
`HashMap`	✓	✓	✓	Sparse, small even for few data
`Immutable`	✓	✓	✓	Cheap copies, sparse
`Indexed`	✓	✓	✓	Fast way to find position by hash code
`SortedArray`	✓	✓	✓	Sparse array with efficient in-order iteration
`Trie`	✓	✓	✓	Fast access and insert, sparse
`Array`	✓	1)	✓	Fast access and insert, dense
`Reverse`	✓	1)	✓	Like `ArrayBackend`, but reverse the data order
`CSR`2) (nested)	3)	3)	✓	Uses little memory, efficient row-major iteration

logical shape, it's cursor logic. For example, a front-end for a hash map contains consecutive key/value tuples. Its shape is described as `@Cflat("*(tuple)->|(key,value)") class MapFlat { ... }`.

Cursor Classes and Iteration. Upon compiling this program, the Java compiler will hand the C♭ expression to our annotation processor, which in turn generates a cursor class. The cursor class can then be used to navigate through the logical view of the front-end:

```
assert new MapFlatCursor().key() == 0;
int val3 = new MapFlatCursor().tuple().tuple().tuple().value();
assert val3 == 3*2 + 1;
```

Additionally, the cursor supports random access with constant time complexity. The following code has the same effect as calling `tuple` N times, but runs in constant time:

```
assert new MapFlatCursor().tuple(3).value() == val3;
```

3.2 Collection of Front-Ends

We provide three basic front-ends: a sequence, a hash map, and a matrix. All of these use back-ends to store their data.

The `Sequence` is the simplest collection in our case study, implementing an ordered collection of elements. The sequence implements Java's `java.util.List` interface, meaning that it can be used as a drop-in replacement for Java's lists.

We implement the `java.util.Map` interface in a class called `MapFlat` that uses a back-end to store its data. The logical representation of the map is a sequence of key/value pairs, expressed by the C♭ expression `*(hash)` $\rightarrow\|$ `(key, value)`. If a key/value pair k/v where k`.hashCode()` $= h$ is to be inserted, the map will attempt to store k at the location `hash`$\times h$`.key` and v at the location `hash`$\times h$`.value` (the shorthand `hash`$\times h$ denotes h repetitions of `hash`). If this location is not empty, but the key that is already inserted does not equal k, there is a hash collision. The map resolves collisions by linear probing: the pair k/v will be inserted at the next location that does not yet contain a key/value pair. When accessing a value by key, the algorithm has to be aware of this particular strategy and potentially iterate in a similar fashion to reach the value.

The `Matrix` class uses a nested back-end to store its two-dimensional data.

3.3 Collection of Back-Ends

We provide a number of back-ends, overviewed in Tab. 2. As expected, all possible combinations of back-ends and front-ends are legal, but not all are memory-efficent.

There are only five operations that every back-end needs to implement. They are:

(1) `T get(final int i)`: access a value at location `i`.

(2) `Backend<T> set(final int i, final T x)`: set a value at location `i`.
(3) `Backend<T> clearAll()`: delete all values.
(4) `int maxIdxOverapproximation()`: return an overapproximation for the latest position that does contain a non-null value.
(5) `Backend<T> emptyCopy()`: return a similar back-end with no values contained (but apply internal size hints, etc, where applicable).

Because many algorithms can be expressed by only relying on these five operations, back-ends come with a number of extra methods (like range-updates, and -queries, iterators, etc.) that have a default implementation. That means that an implementer of a new back-end does not *need* to implement these herself. However, it often makes sense to provide custom implementations, where these can outperform the defaults. Other methods provided by back-ends and used in *e.g.,* the implementation of the `Matrix` class are explained in Sec. 4.1.1.

Switching Representations. All methods that modify a back-end have return type `Back-end<T>`. This makes it easy for a back-end to return a different back-end object on modifications. This may be a back-end of a different kind (*e.g.,* moving from sparse to dense at a certain threshold) but not necessarily, *e.g.,* an immutable back-end will return a copy of itself with the requested change.

4 EVALUATION

We evaluate C♭ through a prototype implementation in Java. We use it to build a number of collections. Using these, we demonstrate the following claims:

C1. C♭ data structures **can have low overhead** compared to Java implementations.
C2. C♭ collection implementations are **straightforward** due to abstracting away layout concerns.
C3. C♭ cursors are **expressive** enough to implement *full-featured* collections.
C4. C♭ data structures **can be tuned** to specific use cases by using different storage implementations, offering both size-over-performance and performance-over-size tradeoffs.

We demonstrate C2 by comparing the algorithms implemented in the C♭ versions with the ones in Java's standard library. We demonstrate C3 by implementing two data structures, lists and hash maps, with the same (large) interface as their corresponding JDK implementation. We demonstrate C4 by showing how different benchmarks benefit from picking the right back-end implementation – and how our data structures can, in these cases, outperform the Java collections. The data structures that we implement are listed in Tab. 3.

Table 3. The use cases and the claims they demonstrate. The matrix does not demonstrate C3 as a full-featured matrix library would be out of scope for the evaluation.

Name	C♭ Expression	Claims	§
Sequence	`*(next)` → `elem`	1–4	Sec. 4.1.1
Hash Map	`*(hash)` →‖`(key, value)`	1–4	Sec. 4.1.2
2D Matrix	Nested Sequences	1, 2, 4	Sec. 4.1.3

4.1 Expressiveness and Ease of Use/Complexity

To our knowledge, C♭ is sufficiently expressive to implement tree shaped data structures. Using C♭ front-ends, more complicated data structures like graphs can be implemented for instance by using the standard adjacency list or adjacency matrix representations.

In this section, we provide qualitative evidence for C2, that using our C♭ implementation can make implementation of data structures easier. This claim's correctness is hard to prove, generally, so we show how the implementations we implement differ in crucial parts from the implementations in Java's standard library. Java's standard library is highly optimised, often ignoring standard software engineering practise by trading code readability for performance—a reader should keep this in mind when comparing C♭ implementations with Java implementations.

```
public boolean add(T e) {
  ensureCapInternal(size + 1);
  data[size++] = e;
  return true;
}
void ensureCapInternal(int minCap) {
  if (data == EMPTY_DATA) {
    minCap =
      Math.max(DEFAULT_CAPACITY,
               minCap);
  }
  ensureExplicitCap(minCap);
}
void ensureExplicitCap(int minCap) {
  // overflow−conscious code
  if (minCap - data.length > 0) {
    grow(minCap);
  }
}
```

```
void grow(int minCap) {
  // overflow−conscious code
  int oldCap = data.length;
  int newCap = oldCap
        + (oldCap >> 1);
  if (newCap - minCap < 0) {
    newCap = minCap;
  }
  if (newCap - MAX_ARRAY_SIZE > 0) {
    newCap = hugeCap(minCap);
  }
  // minCap is usually close to
  // size, so this is a win:
  data = Arrays.copyOf(data, newCap);
}
```

Fig. 6. The add implementation of `java.util.List` needs to grow the internal data array. The C♭ implementation in Fig. 2 does not need to concern itself with that complexity.

In this section, we also provide evidence for C3, that using C♭ permits implementation of full-featured collections, meaning in our Java-context: collections that implement complete interfaces from Java's standard library.

C♭ front-ends in our implementation rely on back-ends that resemble arrays that grow dynamically with the data stored into them (additional non-array-like back-ends would make sense for some use cases, see § 6).

4.1.1 Case Study: Sequence. An excerpt of the class `Sequence`, our list implementation, was shown in Fig. 2. Java lists present their contents with a sequential interface with both the ability to access data by index, as well as via iterators, etc. Using the C♭ expression `*(next)` ⟶ `elem`, the sequence is just a thin wrapper on top of a back-end.

The sequence starts by declaring its simple cursor logic and internal data that it contains.[4] To append a value at the back of the sequence (an operation we benchmark in § 4.2.1), the data is inserted into the backend using the `tail` cursor, and the cursor is advanced one step. For comparison, consider the `ArrayList.add` method from Java 8 (included in Fig. 6 for convenience), where add looks very similar, but also needs to call code to re-grow the internal data in case that's needed to host the new datum. This turns out to be much more complex than the primary task of the method, *i.e.,* adding an element. In the C♭ implementation, this is handled automatically by the back-end, making the implementation of add simpler.

Therefore, even a very simple data structure like a sequence can be simplified, by relying on our implementation of C♭. The full class is 136 SLOC and extends the abstract class `List` (28 methods).

4.1.2 Case Study: Hash Map. Our implementation is very concise: `MapFlat::put`, perhaps the most complex method, is 19 lines of code long, and calls a single helper method that is one line long.

[4]We make slight changes to all code presented in the paper to fit the page layout, like removing `public`/`private` access modifiers, shortening method names, etc.

```
private static double dot(
            final Backend<Double> a,
            final Backend<Double> b) {
    final MutableDouble ret = new MutableDouble(0.0);
    b.joinInner(a, (y, x) -> ret.x += x*y);
    return ret.x;
}

public Matrix multiply(Matrix b) {
    if (this.cols != b.rows) {
        throw new ...;
    }
    Matrix res = new Matrix(
       this.rows, b.cols, (NestedBackend<Double>) this.backend.emptyCopy());
    this.backend.foreachNonNull(i ->
            b.backend.foreachColNonNull(j -> {
                res.put(i,j, dot(getRow(i), b.getCol(j)));
    }));
    return res;
}
```

Fig. 7. A Matrix implementation in C♭ that uses iteration methods on back-ends. Back-ends can specialise these methods for better performance.

For reference, `HashMap.put` is 169 SLOC (39 SLOC for its implementation and a further 130 SLOC for the used helper methods[5]). Note that there are at least two underlying reasons for this difference: first, `java.util.Hash-Map` manages its layout internally and needs extra code to do that (it uses an array of bins to store its data) while this complexity is abstracted away when using C♭ back-ends. Second, `java.util.HashMap` is highly optimised, it would likely require fewer lines of code if it was less optimised[6]. Consequently, we do not claim that this difference in length is purely due to using C♭, just that the C♭ implementation is concise. The full class is 87 SLOC and extends `AbstractMap` (25 methods). Note that both the map and the sequence are able to make use of inheritance.

4.1.3 *Case Study: Matrix.*

The implementations in Java's standard library are not necessarily representative of the code most programmers write. As these classes are relied on by many users, they are heavily optimised and do not necessarily follow established coding practice. To compare C♭ to less optimised Java code, we implement two simple matrix classes which support matrix multiplication: one using C♭ and one written in pure Java. We compare these two implementations here, and evaluate their multiplication performance in Sec. 4.3.3.

The C♭ `Matrix` represents a matrix of **`double`** values using a nested back-end, which contains rows of matrix data, making the front-end view the data in a row-major fashion (this does not imply that data must have this layout, as explained later in the performance evaluation). We implement the standard matrix multiplication algorithm that traditionally has the asymptotic complexity $O(N^3)$ for $N \times N$ matrix multiplication. In C♭, however, the complexity of that algorithm depends on the implementation of the back-end used to store the matrix' data.

[5]`HashMap.resize`: 71 SLOC, `HashMap.treeifyBin`: 18 SLOC, `HashMap.putTreeVal`: 41 SLOC.
[6]For instance, `java.util.HashMap` will turn bins that contain many values into trees after a certain size, to mitigate performance problems from bad hash code implementations.

The algorithm for multiplication, see Fig. 7, is structurally similar to what a programmer in traditional Java would write, but essentially replaces **for** loops with iteration methods on back-ends. The key feature here is that operations like `joinInner`, `foreachNonNull`, and `foreachColNonNull` can be overridden by the back-end implementations. The `joinInner` operation takes two back-ends, and executes an *inner join* (like in a relational data base) by calling the passed anonymous function for each pair of non-null values x, y that are stored at the same location in the two back-ends. The `foreachNonNull` and `foreachColNonNull` operations call the passed anonymous function with each row- or column-index of rows or columns that store non-**null** data.

The `dot()` operation relies on `joinInner()`. If this method is implemented by a back-end to have a complexity in the order of the number *non-null values*, rather than the number of rows/columns of the matrix, multiplication can be much faster. Consider calling `joinInner()` to join two `ArrayBackend`s, each containing V values distributed over N slots, where $V \ll N$: the algorithm has to go through both back-ends to check whether they contain a non-null value at a given index and, if so, call the passed anonymous function, using an $O(N)$ algorithm. In comparison, two `SortedArrayBackends` (see Fig. 4d) can be joined much more efficiently, as the join algorithm just has to compare the already ordered position arrays of both back-ends (see Sec. 2.4.1 and Fig. 4d). A programmer of a matrix needs no knowledge of any of this, as a user of the matrix can try which representation performs best for her specific use case.

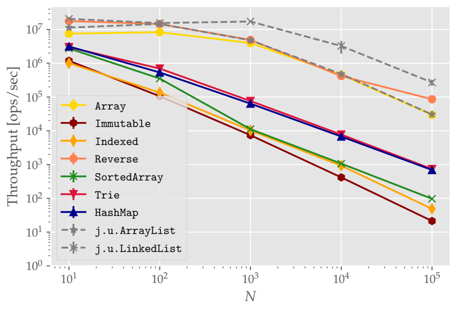
4.2 Performance Comparison with `java.util` Collections

We show that C♭ data structures can get near Java's standard library (but are slower in most cases) for common operations, when provided the right back-end. We compare the classes `java.util.ArrayList`, `java.util.LinkedList` and `java.util.HashMap` to the combinations in Tab. 2). These are not strawman collections: Java's `Map` interface, and the `HashMap` were added to Java 1.2, in 1998 and have been updated, tuned and optimised for ≈20 years, while our backends have not been optimised as much.

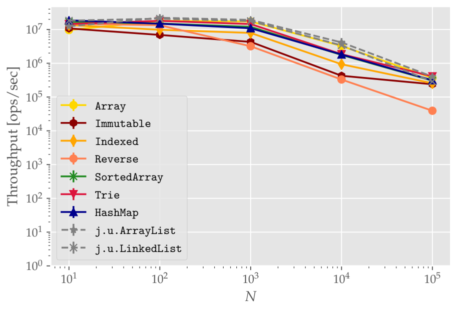
We use micro benchmarks running the most important methods. The back-end that works well for the list in most cases is the `ArrayBackend`, which gives the our implementation performance characteristics very close to `java.util.ArrayList`. For the hash map, the back-end called `TrieBackend` works best in most cases. This back-end splits the integer position into its four consecutive bytes and uses those as indices into a 256-ary tree. The back-end is relatively space-inefficient when it contains only few elements but works better the more data is stored.

4.2.1 List Performance. We compare the performance of our Sequence and Java's linked- and array-list implementations by benchmarking the time it takes to add a value to the list (at the front and at the back). We also measure the time it takes to access a list at a random location.

Add at the front of a list. This benchmark, whose results are shown in Fig. 8a, inserts a single value at the beginning of a list of length N, increasing its length by one. This use case is a pathological use of `java.util.ArrayList`, and most of our back-ends, as all the data in the list has to be moved to the right by one. Java's linked list (`java.util.LinkedList`) stands out: inserting at the beginning of the linked list has constant time complexity, as the list only needs to allocate a new node and link it into the list. `ReverseBackend` is a back-end that contains another back-end (an `ArrayBackend` in this case) *with reversed order.* This means that inserting at the beginning of a Sequence backed by a `ReverseBackend` is a constant time operation – but appending to the list would instead be expensive (*c.f.* Fig. 8b). We think that a backend based on a linked list would perform comparable (but with some overhead) to `java.util.LinkedList` but currently don't have such an implementation. Compared to the `java.util.ArrayList`, and to the backend implementations we have, the `ReverseBackend` is better or comparable for all tested lengths.

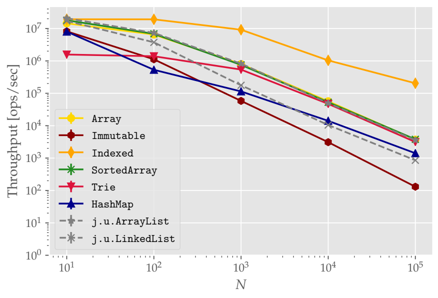


(a) Inserting at the front of a list, see Sec. 4.2.1.

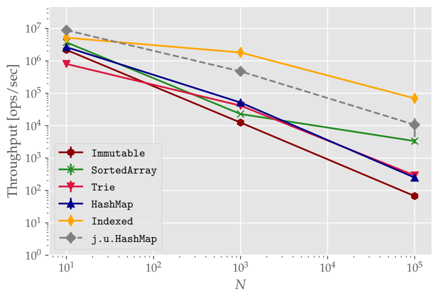


(b) Appending at the end of a list, see Sec. 4.2.1.

Fig. 8. Inserting data at the front or back of a list. At the back, our implementation is comparable to java's collections, while at the front, none of our back-ends can reach the linked list (a back-end encapsulating a linked list would likely work well here).



(a) Checking whether a non-existing value is in a list, see § 4.3.1.



(b) Checking whether a non-existing value is in a map, see § 4.3.1.

Fig. 9. A single back-end, IndexedBackend, can be used to speed up operations on both the list *and* the map. This back-end maintains an index of the values it stores, making it cheap to find their location.

The benchmark in Fig. 8b appends a single value at the end of a list. Most lists deal well with this use case, as the append operation has constant time complexity in these cases. The back-end that stands out negatively is the ReverseBackend, which needs to shift all the elements back by one to fit the new element.

Access random location in a list. The benchmark in Fig. 10 measures the the time it takes to access a random value in a list of length N using the same sequence of locations for each data structure. Java's array list wins, with performance slightly better than our sequence with the ArrayBackend.

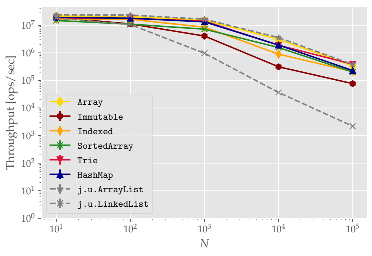
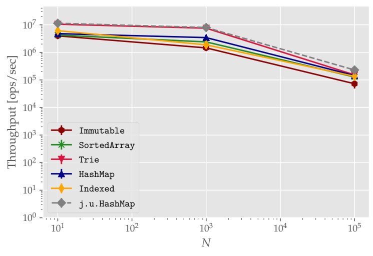


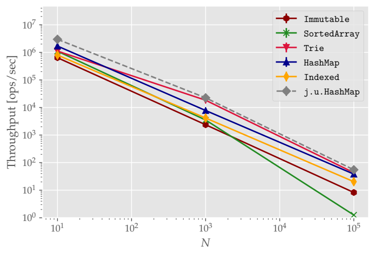
Fig. 10. Accessing a random location in a list, see Sec. 4.2.1.

4.2.2 Hash Map Performance. We compare the performance of our map to Java's `HashMap` by looking at inserting and accessing data.

Access random existing string key. The benchmark in Fig. 11a accesses a random key in a map containing N key/value pairs where the keys are strings. Using our map, together with the `TrieBackend`, we are able to reach performance close to Java's hash map.



(a) Accessing a string key in a map data structure, see Sec. 4.2.2.



(b) Inserting string keys into a map data structure.

Fig. 11. Map access and insertion.

Insert non-existing string key. The benchmark in Fig. 11b inserts N key/value pairs into an initially empty map data structure. The `TrieBackend` back-end starts out slower than Java's hash map, but its performance gradually climbs to match it for larger values of N. We believe the slow performance for lower values of N to be due to the `TrieBackend` requiring large amounts of memory (it is a 256-ary tree). A data structure that might work better would be a mutable hash-array mapped trie [Bagwell 2001].

4.3 Tuning Performance by Changing Physical Representation

We now show how using different back-ends can yield data structures that either perform better for a specific operation, or that require less memory.

4.3.1 Reducing Computation Time. In this section, we show how using *one* single specialised back-end class can speed up operations on *two* different data structures.

Specifically, we look at the performance of the list's `indexOf(..)` and the map's `containsValue(..)` operations, which both check whether a given value is contained in the collection. Both of these data structures traditionally implement these operations with linear complexity – they have to iterate over all contained data to check for value membership.

With C♭, we can improve these operations using a single backend class: the `IndexedBackend` stores its data in hash map, but it also maintains a second hash map that maps the hashes of its values to the value's locations. This means that this back-end supports searching the location of a value efficiently.

We benchmark this back-end by initialising a collection with N values and searching for the index of a *non-existing* value using the `indexOf` method. The C♭ collections using the `IndexedBackend` outperforms the corresponding Java collections by an order of magnitude for large values of N (see Figures 9a and 9b for graphs).

4.3.2 Reducing Memory Usage. Just like we can use the choice of back-end to save computation time, we can choose a back-end to cut down on the memory required to store data. This can make C♭ front-ends an acceptable choice where their Java-equivalent wouldn't be. For example, Java's lists

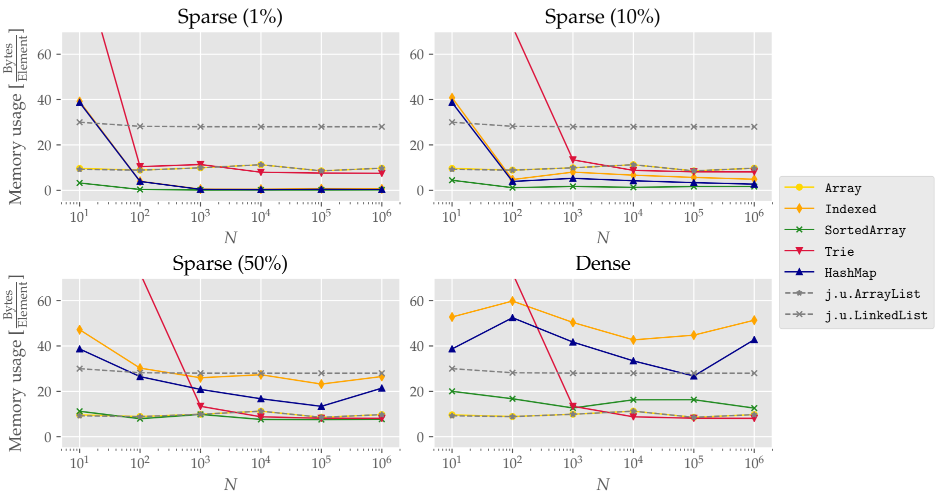


Fig. 12. *Sequence sparseness vs. length vs. memory usage.* Memory savings increase with amount of **null** values. The y-axis shows the amount of memory used *per non-***null** *value*.

are rarely used for sparse data – Fig. 12 offers explanation why: the memory required per non-null element is too high.

Fig. 12 shows the memory required to store a list of length N with various back-ends. Using a back-end that does not require space for **null** elements like SortedArrayBackend, we can reduce the memory cost without modifying the list. Such lists can be useful to, *e.g.*, store rows in a sparse row-major matrix.

Sparse lists are relatively simple, hash maps are more complex. Using a sorted array back-end, we can nearly half the memory compared to Java's HashMap (*c.f.* Fig. 13).

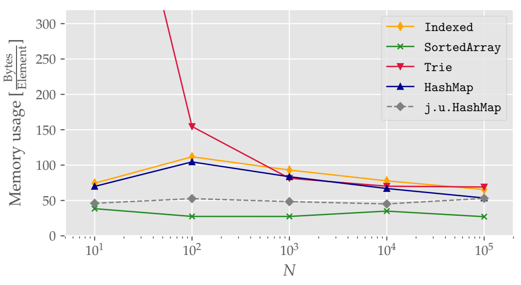
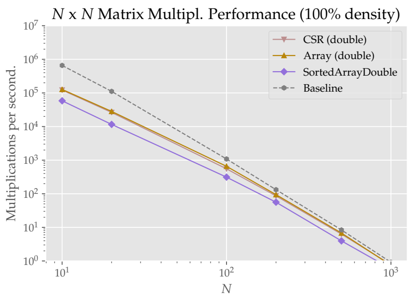


Fig. 13. The memory a map with N key/value pairs requires per inserted key/value pair. Omitting data for ImmutableBackend, as the used data structure [Steindorfer and Vinju 2015] does not report its size in bytes.

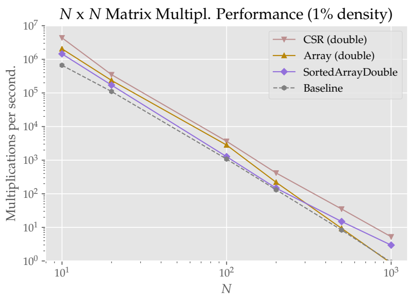
4.3.3 *Adaptive Matrix Multiplication.*

We explained what nested back-ends are in Sec. 2.4.2, but we have yet to investigate their performance. As a simple experiment, we benchmark the multiplication of sparse ($\approx$ 1% of values are defined, the rest is empty) and of dense square matrices, with sizes varying from 10×10 to 1000×1000 and compare it to a Java-only baseline. The baseline is simply a matrix that arranges all of its data in a **double**[] array in row-major or column-major order (it uses a boolean field to track whether it is in row- or column major representation). The multiplication algorithm is the same, but uses for-loops in an idiomatic way where the C♭ implementation uses iteration methods.

The nested back-ends we use are the CSRBackend, as a representative of a nested back-end that does not represent its internal rows as separate back-ends (sparse, see Sec. 2.4.2), a nested back-end constructed from an ArrayBackend (dense, see Fig. 5a) containing ArrayBackend rows, and a nested back-end constructed from a SortedArrayBackend containing SortedArrayBackend rows (sparse, a non-nested sorted array is depicted in Fig. 4d). All the back-ends used have been specialised



(a) The baseline implementation is significantly faster for small matrix sizes, C♭ implementations improve with larger matrices, almost catching up at $N = 1000$.



(b) Using a `CSRBackend`, we can tune multiplication performance for sparse matrices (for $N = 1000$, the speedup over the baseline is 5.7×).

Fig. 14. Performance of matrix multiplication for selected back-ends, and comparing sparse vs. dense matrices.

manually to store primitive **`double`** values internally, rather than `java.lang.Double` objects, to make the comparison to the baseline implementation fair. Additionally, as all our nested back-ends are row-major by design, but matrix multiplication works better with a column-major storage on the right hand side, we added a back-end `TransposedBackend` to our framework that wraps a back-end and translates all accesses to coordinates (i, j) to accesses on the inner back-end at locations (j, i) and use that wrapper in the experiment. This means that we can implement column-major storage of a matrix wholly at the back-end level, without the matrix having to implement this feature–making the C♭ matrix somewhat simpler than the baseline implementation.

For dense data, the results show that the Java baseline implementation performs significantly better than our C♭ implementation for small matrices; for large matrices, the CSR back-end and the array back-end both catch up. We believe the difference for small matrices to be due to pointer-chasing effects (our matrix contains a back-end which contains more data structures before the actual data is being reached, while the Java-only matrix contains only a **`double`**`[]` array). The overhead of these effects, relative to the amount of work, gets smaller with increasing work (*c.f.* Fig. 14a).

For sparse data, our results consistently show our CSR back-end outperforming the Java baseline as well as the sorted array back-end. This is not surprising—using the CSR back-end, the algorithm simply has less work to do. The key takeaway is that a user does neither need to fully understand the algorithm, nor the back-ends involved, she merely needs to find a back-end that happens to work well in practice.

4.4 Conclusion of Evaluation

We have provided evidence for the claims presented at the beginning of this section:

C1 *C♭ data structures can have low overhead compared to Java implementations.* We have shown, in § 4.2, that execution time for important methods of the `Sequence` and `MapFlat` implementations are comparable (yet often not quite as good; the lager the collections are, the better our implementations tend to perform) to Java's `ArrayList` and `HashMap`, respectively.

C2 *While a departure from the traditional way of implementing collections, C♭ code is straightforward.* We have shown, in § 4.1, that our sequence, map, and matrix are concise and high-level.

C3 *C♭ implementations can be as full-featured as implementations that access the physical representation directly.* We have two data structures (`Sequence` and `MapFlat`) implementing Java's `List` and `Map` interface, respectively. Both of these interfaces have significant size (the list interface has 28 methods, the map interface has 25).

C4 *C♭ collections are tunable.* We have shown, in § 4.3, that we can use specialised back-ends to improve the performance of operations that Java's `Lists`, and `HashMap` do not excel at using the example of testing whether they contained a value. We also showed how a specialised back-end speeds up multiplication of sparse matrices (see Fig. 14a).

5 RELATED WORK

Synthesis of Data Structures. Work on data representation synthesis [Hawkins et al. 2011] produces specialisable data structure implementations from high-level descriptions. The work can synthesise containers that can hold components that are indexed with a number of index keys. Each unique combination of index keys stores an associated value record. In comparison, while representation synthesis produces full-blown data structures, C♭ abstracts away the spine of data structures but lets a programmer implement the logic. C♭ is, in our view, more expressive, but also demands more code from programmers. In a similar vein, the Koloboke-compile project produces Java-hash map implementations and can produce sophisticated algorithmic optimisations. For example, a map that does not support removing keys can use a more efficient algorithm than one that does. C♭ is more general – users can implement new front-ends without modifying the compiler.

Generating (parts of) data structures is established practice in today's software engineering. For example in Java there are libraries that provide data structures that are specialised to all combinations of primitive values for their type parameter. This can lead to reduced memory requirements and increased cache performance. One such example is Trove [2018], that provides lists, maps, etc. that specialise their key/value parameters. Trove uses code generation to produce the repetitive code for these specialisations. Trove implements the layout of a data structure manually and specialises it to different key/value types. This is orthogonal to C♭ and the Trove approach might also be useful to specialise back-ends.

Closely related work is work on *just-in-time data structures* [De Wael 2015]. In this work, a user can implement a data structure in a number of different ways and get automatic switching between data representations. This is a feature very similar to our tunable data structures. The work shows that such data structures that adapt their representation on the fly can significantly improve the performance of large benchmarks. C♭ differs because a front-end needs to be implemented only once, and is then freely composed with the available back-ends, while the user of just-in-time data structure has to implement all the combinations from scratch.

Adaptive Data Structures. Adaptive data structures themselves are not a new idea and have been used in many contexts, from data bases [Mitra et al. 2013] to parallelism [Sagonas and Winblad 2018]. In C♭, adaptive behaviour can be implemented by switching between back-end representations during program execution.

C♭ is somewhat similar in spirit to work by Bolz et al. [2013] that implement storage strategies that optimise the representation of monomorphic collections at the JIT level. This allows *e.g.,* storing unboxed integer representations, but does not focus on *e.g.,* sparse vs. dense collections. Bolz et al.'s collections use the strategy pattern to interact with the underlying representation, focusing on efficient storage of objects of different types. A collection can change storage strategy over time to respond to changes in the objects it stores. The ability to change storage over time is something we have considered but not investigated for C♭. For example, we could support changing between sparse and dense representation with respect to some threshold value, or change from a back-end that premiers efficient insertion to one that premiers efficient look-up, as data enters a stable state. The unified external interface to back-ends would facilitate this kind of optimisation. Automatically identifying triggering points for changing back-ends, and automatic choice of back-end on a change point are interesting directions for future work.

Language Support for Data Layout. The impact on data layout on performance has been well-studied in the past. Two common techniques are pooling [Franz and Kistler 1998] – placing objects together in memory based on *e.g.,* type, allocation site or profiling information, often in combination with splitting pooling [Chilimbi and Shaham 2006; Curial et al. 2008; van der Spek et al. 2010; Wang et al. 2012]. Lattner et al. [2003; 2005] apply static analysis to C and C++ program to perform allocation in a cache-friendly way, and Calder et. al. [1998] use dynamic profiling information. Aforementioned systems above are all transparent to the programmer. Franco et. al [2017] proposed SHAPES, a high-level programming language where layout parameters are used to pool and split objects in a memory-safe way. Whereas in C♭ "business logic" and storage is separated, SHAPES conflates them, albeit in a tractable way. A SHAPES program may be tuned by changing *e.g.,* how objects in a pool are split or ordered, but cannot change on a fundamental level like changing from array-based to hashmap-based.

Gibbon [Vollmer et al. 2017] compiles large immutable algebraic data types to packed representations that have benefits for operations that operate on, say, all contained values in bulk. It is less flexible than C♭ but being in the compiler means that it can likely produce more optimal code.

6 FUTURE WORK

The current implementation works well for the data structures we chose in § 4. However, the data structures we chose are all degenerate trees: a list, a hash map, a matrix, and a red-black-tree (omitted from the evaluation for brevity). Our current implementation of a red-black-tree has bad performance: the reason is that some operations in trees that are very cheap in "standard" Java can not be implemented in C♭ back-ends easily. Consider the following operation in Java that moves a sub-tree in a binary tree node: `n.left = n.right; n.right = null`.

The same operation in current C♭ would physically move all the memory from the right subtree to the corresponding locations in the left subtree. This requires linear time, $O(N)$, where N is the number of nodes reachable from `n`.

We are aware of two solutions to this issue. The first solution is to build a back-end that tracks move-subtree operations symbolically (rather than executing the expensive physical move). This can be done with no changes to our current implementation, but it puts extra cost on reads. The second solution generates a back-end that mirrors the expression's shape, for every C♭ expression. For a binary tree expression, the internal storage of such a back-end would be binary tree-shaped, with the same cheap move operation as the normal Java implementation. This requires changes to the implementation and a refactoring of the back-end library but would provide performance close to a traditional Java implementation – while still having the ability to benefit from the back-ends presented in § 4 for trees that are not often re-balanced (like red-black-trees that are modified rarely, but read many times).

7 CONCLUSION

We have presented C♭, a novel approach to implementing collections where the implementation of the logical front-end is separated from the back-end storage. This separation keeps the front-end implementation simple and allows the same front-end to be reused with different back-ends as access patterns and non-functional requirements vary between use cases. Similarly, it allows improvements made to the back-end of one collection to have positive effects to other collections using the same back-end. Our prototype implementation shows that C♭ can produce collections with performance that is tunable and fit for most practical uses, and that the front-end implementations can be kept simple without sacrificing expressivity of individual collections. There is still work to be done (*c.f.* § 6), but our preliminary results suggest that C♭ can make front-ends and back-ends work in harmony to efficiently implement full-scale collection libraries.

REFERENCES

Phil Bagwell. 2001. *Ideal Hash Trees.* Technical Report LAMP-REPORT-2001-001. École polytechnique fédérale de Lausanne.

Carl Friedrich Bolz, Lukas Diekmann, and Laurence Tratt. 2013. Storage strategies for collections in dynamically typed languages. In *Proc. of the 2013 ACM SIGPLAN Intl. Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013.* 167–182. DOI: http://dx.doi.org/10.1145/2509136.2509531

Aydin Buluç, Jeremy T. Fineman, Matteo Frigo, John R. Gilbert, and Charles E. Leiserson. 2009. Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks. In *SPAA 2009: Proc. of the 21st Annual ACM Symposium on Parallelism in Algorithms and Architectures, Calgary, Alberta, Canada, August 11-13, 2009.* ACM, 233–244. DOI: http://dx.doi.org/10.1145/1583991.1584053

Brad Calder, Chandra Krintz, Simmi John, and Todd Austin. 1998. Cache-conscious Data Placement. *SIGPLAN Not.* 33, 11 (Oct. 1998), 139–149. DOI: http://dx.doi.org/10.1145/291006.291036

Trishul M. Chilimbi and Ran Shaham. 2006. Cache-conscious Coallocation of Hot Data Streams. In *PLDI '06.* ACM, 252–262.

Stephen Curial, Peng Zhao, Jose Nelson Amaral, Yaoqing Gao, Shimin Cui, Raul Silvera, and Roch Archambault. 2008. MPADS: Memory-pooling-assisted Data Splitting. In *ISMM '08.* ACM, 101–110.

Mattias De Wael. 2015. Just-in-time Data Structures: Towards Declarative Swap Rules. In *Proc. of the 13th Intl. Workshop on Dynamic Analysis (WODA 2015).* ACM, New York, NY, USA, 33–34. DOI: http://dx.doi.org/10.1145/2823363.2823371

Juliana Franco, Martin Hagelin, Tobias Wrigstad, Sophia Drossopoulou, and Susan Eisenbach. 2017. You can have it all: abstraction and good cache performance. In *Proc. of the 2017 ACM SIGPLAN Intl. Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Onward! 2017, Vancouver, BC, Canada, October 23 - 27, 2017.* 148–167. DOI: http://dx.doi.org/10.1145/3133850.3133861

Michael Franz and Thomas Kistler. 1998. *Splitting Data Objects to Increase Cache Utilization.* Technical Report. UC Irvine.

Peter Hawkins, Alex Aiken, Kathleen Fisher, Martin Rinard, and Mooly Sagiv. 2011. Data Representation Synthesis. In *Proc. of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11).* ACM, New York, NY, USA, 38–49. DOI: http://dx.doi.org/10.1145/1993498.1993504

Chris Lattner and Vikram Adve. 2003. *Data Structure Analysis: A Fast and Scalable Context-Sensitive Heap Analysis.* Technical Report. U. of Illinois.

Chris Lattner and Vikram Adve. 2005. Automatic Pool Allocation: Improving Performance by Controlling Data Structure Layout in the Heap. In *PLDI '05.* ACM, 129–142.

Pinaki Mitra, Girish Sundaram, and Sreedish P. S. 2013. Just In Time Indexing. *CoRR* abs/1308.3679 (2013). arXiv:1308.3679 http://arxiv.org/abs/1308.3679

Chris Okasaki. 1996. Functional data structures - Advanced Functional Programming. Springer Berlin Heidelberg, 131–158.

Konstantinos Sagonas and Kjell Winblad. 2018. A contention adapting approach to concurrent ordered sets. *J. Parallel and Distrib. Comput.* 115 (2018), 1 – 19. DOI: http://dx.doi.org/https://doi.org/10.1016/j.jpdc.2017.11.007

Michael J. Steindorfer and Jurgen J. Vinju. 2015. Optimizing Hash-array Mapped Tries for Fast and Lean Immutable JVM Collections. In *Proc. of the 2015 ACM SIGPLAN Intl. Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2015).* ACM, New York, NY, USA, 783–800. DOI: http://dx.doi.org/10.1145/2814270.2814312

Development Team of Trove. 2018. GNU Trove: High performance collections for Java. https://bitbucket.org/trove4j/trove. (2018). BitBucket repository, accessed April 22 2018.

Harmen L. A. van der Spek, C. W. Mattias Holm, and Harry A. G. Wijshoff. 2010. Automatic Restructuring of Linked Data Structures. In *LCPC'09.* Springer, 263–277.

Michael Vollmer, Sarah Spall, Buddhika Chamith, Laith Sakka, Chaitanya Koparkar, Milind Kulkarni, Sam Tobin-Hochstadt, and Ryan R. Newton. 2017. Compiling Tree Transforms to Operate on Packed Representations. In *31st European Conference on Object-Oriented Programming (ECOOP) (LIPIcs)*, Vol. 74. Schloss Dagstuhl, Dagstuhl, Germany, 26:1–26:29. DOI: http://dx.doi.org/10.4230/LIPIcs.ECOOP.2017.26

Zhenjiang Wang, Chenggang Wu, Pen-Chung Yew, Jianjun Li, and Di Xu. 2012. On-the-fly Structure Splitting for Heap Objects. *ACM TACO* 8, 4 (2012), 26:1–26:20. DOI: http://dx.doi.org/10.1145/2086696.2086705