Collection Skeletons: Declarative Abstractions for Data Collections

Björn Franke
bfranke@inf.ed.ac.uk
School of Informatics
University of Edinburgh
Edinburgh, Scotland, United Kingdom

Zhibo Li
zhibo.li@ed.ac.uk
School of Informatics
University of Edinburgh
Edinburgh, Scotland, United Kingdom

Magnus Morton
magnus.morton@huawei.com
Huawei Research Centre
Edinburgh, Scotland, United Kingdom

Michel Steuwer
michel.steuwer@ed.ac.uk
School of Informatics
University of Edinburgh
Edinburgh, Scotland, United Kingdom

Abstract

Modern programming languages provide programmers with rich abstractions for data collections as part of their standard libraries, e.g. Containers in the C++ STL, the Java Collections Framework, or the Scala Collections API. Typically, these collections frameworks are organised as hierarchies that provide programmers with common abstract data types (ADTs) like lists, queues, and stacks. While convenient, this approach introduces problems which ultimately affect application performance due to users over-specifying collection data types limiting implementation flexibility. In this paper, we develop *Collection Skeletons* which provide a novel, *declarative* approach to data collections. Using our framework, programmers explicitly select *properties* for their collections, thereby truly decoupling specification from implementation. By making collection properties explicit immediate benefits materialise in form of reduced risk of over-specification and increased implementation flexibility. We have prototyped our declarative abstractions for collections as a C++ library, and demonstrate that benchmark applications rewritten to use Collection Skeletons incur little or no overhead. In fact, for several benchmarks, we observe performance speedups (on average between 2.57 to 2.93, and up to 16.37) and also enhanced performance portability across three different hardware platforms.

ACM ISBN 978-1-4503-9919-7/22/12…$15.00
https://doi.org/10.1145/3567512.3567528

CCS Concepts: • **Software and its engineering → Abstraction, modeling and modularity**; Feature interaction.

Keywords: Containers, collections, data structures, properties

ACM Reference Format:
Björn Franke, Zhibo Li, Magnus Morton, and Michel Steuwer. 2022. Collection Skeletons: Declarative Abstractions for Data Collections. In *Proceedings of the 15th ACM SIGPLAN International Conference on Software Language Engineering (SLE '22), December 06–07, 2022, Auckland, New Zealand.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3567512.3567528

1 Introduction

Collections of data items are central to many fundamental algorithms, and find use in all applications storing, processing and retrieving data. Therefore, it is not surprising that collections have been at the heart of Computer Science research since the inception of the discipline. From lists, stacks, queues through trees and maps to union/find data structures, a great number of *abstract data types (ADTs)* [17] and concrete data structure implementations along with efficient algorithms for the organisation and efficient retrieval of data have been developed over the years [9].

While individual data structures and algorithms for general data collections are well understood, there is less consensus about the relationship, or more specifically, the *hierarchy*, of different kinds of collections. The designers of collection abstractions for different programming languages have taken quite different approaches to organising collections in object-oriented class hierarchies. For example, the *C++ Standard Template Library (STL)* [24], the *Java Collections Framework (JCF)* [21], and the *Scala Collections Framework (SCF)* [22] capture essentially the same collections in different ways. This is because existing class hierarchies and design patterns for collections are *operation centric*, where inheritance relationships dictate the structure. Furthermore, specific non-functional requirements like the prescribed algorithmic complexity of certain operations in the C++ STL leave little choice when implementing STL containers. Despite superficial similarities between the collection hierarchies in the C++ STL, the JCF and the SCF, there are major differences between the frameworks and any programmer familiar with the fundamental concepts of one of the frameworks would need to spend significant effort to familiarise themselves with the other frameworks before becoming confident in their efficient use [6]. To illustrate this we refer to the JCF, where a *Stack* extends a *Vector*, which in turn implements a *List*; while a *LinkedList* implements both a *List* and a *Deque* interface. Figure 1 illustrates the hierarchies for both data collections, which appear non intuitive in terms of semantic properties of the collections represented. Why would a stack be a vector or a list?

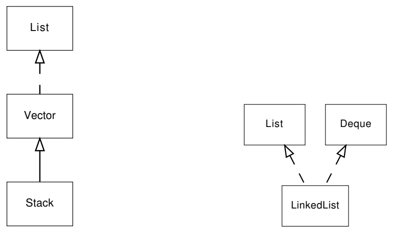


Figure 1. Excerpt from the Java Collections Framework, showing the inheritance hierarchy for a Java Stack (left) and LinkedList (right).

In this paper, we develop a novel approach to providing user-facing abstractions for data collections. In our *Collection Skeletons* framework, programmers do not instantiate a collection of a certain class (e.g. `std::list<int>` for a list of integers), but instead, they explicitly specify properties that the collection must provide. The **key idea** is to entirely decouple specification from implementation of collections. Instead of abstract or concrete data types for their data collections, programmers *only* specify the properties their collections are relying on, thus giving the Collection Skeletons framework the flexibility to select *any* concrete implementation that provides the requested properties. For this, we distinguish between two kinds of properties concerned with 1) the *semantics* of collections, i.e. the expected behaviour, and 2) their *interfaces*, i.e. the available methods to access functionality. Although there are properties that do not belong to either semantics or interfaces, e.g. memory efficiency of a data collection, those *non-functional* properties are beyond the scope of this paper and are subject of our future work. Unlike existing collection frameworks, Collection Skeletons do not attempt to fit different kinds of collections into a hierarchy, but instead we provide a single versatile and parameterisable collection type.

Semantic properties refer to the expected behaviour of collections. For example, whether or not a collection is allowed to store duplicate entries, or if it is restricted to unique data items, is a semantic property. *Interface properties* relate to the provision of specific functions to interact with the collection, e.g. whether or not a *split-by-value* operation for splitting collections around a pivot element is provided. We discuss these properties in detail in section 2.

We have prototyped our novel Collection Skeletons framework as a C++ library. For its evaluation, we have rewritten several benchmark applications exercising collections of different kind and nature, and executed them on three different hardware platforms (6-core Intel NUC 10, 72-core Intel Gold 6154 and 4-core Oracle Cloud Arm server). This enables us to measure any potential overheads introduced by our abstractions. In fact, we demonstrate that our Collection Skeletons introduce only negligible runtime overhead, and deliver performance improvements on several occasions. This is because our framework enables us to select an implementation different from the collection used in the original code base, which results in higher performance. We also show that the best possible implementation choice for a given collection is program- and platform-dependent. This is where the flexibility of Collection Skeletons is of clear benefit: Concrete implementations can be flexibly swapped out without modification of the user's application code.

1.1 Motivating Example

Consider the linked-list example in Figure 2. Listing 1 shows a code snippet with a loop traversing a user-defined linked list as it would be commonly found in C code. The user specifies a concrete implementation, here a single-linked dynamically allocated list using a user-defined data type. The traversal loop performs pointer-chasing to move from element to element. The choice of implementation *implicitly* defines semantic properties, e.g. the order in which elements are stored and the possibility for duplicate elements. Using the C++ STL as shown in Listing 2 the user utilises a `std::list` collection data type, and the explicit pointer-chasing from Listing 1 is replaced by a range-based for loop. This notionally introduces a *list* abstract data type, where the concrete list implementation is hidden behind an operational list interface. The user relies on the properties provided by the *list* ADT as defined in the C++ STL. In contrast, using the Collection Skeletons approach shown in Listing 3, the user makes explicit the properties (e.g. storage order, sequential accessible) they request for their collection. We request a *Seq* property since we subsequently iterate over the elements one by one and we choose *variable* length since the number of elements contained in the collection is not statically known, while we may rely on a specific storage *order* and allow *duplicates*, possibly because part of an algorithm not shown in the examples requires these properties.

We refer to the *Seq* property as an *interface* property since it is linked with the provision of operational facility through interface functions to iterate one by one over the collection.

```
typedef struct nodes{
  nodes *next;
  int data;
};

for(;node;node=node->next)
     {
  node->data += 1;
}
```

Listing 1. User-defined list in C.

```
std::list<int> nodes;




for(auto& i : nodes) {
  i += 1;
}
```

Listing 2. List using the C++ STL.

```
Collection<int, Seq, Dup, Variable
    , Ordered> c;




for(auto& i : c) {
   i += 1;
}
```

Listing 3. Collection Skeletons.

Figure 2. Motivating example showing the evolution of a linked list collection and its traversal expressed in C, C++ STL, and eventually using Collection Skeletons. While the C programmer employs a user-defined linked list data structure, the C++ STL offers a pre-defined list collection. In contrast, using Collection Skeletons a programmer requests collection by explicitly requesting the properties they rely on for maximal implementation flexibility.

Sequential accessible has also indicated that the collection need to be ordered at the storage level.

Our motivating example shows Collection Skeletons provide a convenient abstraction to collection data types, which avoid tedious and non-intuitive hierarchies of collections, but instead equip the users to specify exactly the properties they need for their application. We show in our evaluation in section 4 how this abstraction incurs only negligible overhead.

Overall, this paper makes the following contributions:

1. We develop a novel declarative approach to specifying data collections, exposing individual properties to be specified (rather than a pre-packaged sets of properties like in ADTs),
2. we identify a set of useful semantic and interface properties, which capture the key aspects of data collections programmers care about, and
3. we evaluate a prototype C++ library implementation of our Collection Skeletons framework against legacy benchmarks rewritten to make use of our new abstraction, and demonstrate negligible performance impact across three different hardware platforms.

2 Collection Skeletons

Collection Skeletons compete with the concept of Abstract Data Types for the specification of data collections. ADTs are mathematical models for data types, which specify user-facing signatures and semantics of operations on a data type. Formally, ADTs are defined by either axiomatic semantics or operational semantics of an abstract machine. Specific properties of ADTs are thus expressed through the semantics of the operations it provides, e.g. for a *stack* we might expect a definition that captures that a *pop* operation following immediately after a *push x* returns the value *x*, and the state of the *stack* to be the same as it was before. In this sense, ADTs *implicitly* define properties through semantics of its operations. In addition, there is no notion of relationship between ADTs. For example, users might perceive *stacks* and *queues* are related collection data types that only differ in their data retrieval order, but their respective ADTs do not attempt to establish this similarity.

Collection Skeletons follow a different approach, where all collections are derived from a single, versatile *Collection* type by parameterisation. The parameters to this *Collection* archetype are semantic and interface properties, respectively. Semantic properties relate to e.g. whether data elements are unique or if duplicate entries are allowed, whereas interface properties specify available access functions for the programmer to interact with the collection.

We propose eight preliminary groups of properties to help model the Collection Skeletons in our prototype library. Properties belonging to the same group exhibit similar features or a twofold symmetrical dichotomy. By combining the properties following proper rules, different data collections can be defined. Table 1 presents the eight groups of properties, their definitions, and the API parameters which are abbreviations to the properties for convenient use, in our library. We will discuss the API parameters in section 2.

Using these *declarative* abstractions for data collections, it is straightforward to define a data collection by specifying only the desired properties. We integrate the property-based model with modern C++ programming practice, making the learning curve for end users as smooth as possible. Our implementation employs C++ template meta-programming, in which a collection can be defined as:

$$\texttt{Collection<T,}P_1\texttt{,}P_2\texttt{,...}P_n\texttt{,(F...)>}$$

where T is the elementary data type of the collection and P_1 to P_n are parameters from API column of Table 1. F... are optional parameters that maybe specified by the user when necessary, which we will later discuss in detail. Using rule based combinations of properties, different data collections, including the standards collection ADTs from other collection frameworks, can be defined.

Table 1. Groups of properties, their definitions and API usage

Semantic property	Definitions	API Parameters
Uniqueness	$val(x) \neq val(y) \forall x, y \in \mathsf{c}$ for Unique; otherwise for Duplicated	Unique Dup
Circularity	$val(pos(i' + i)) = val(i) \forall i \in \mathsf{c}$ and $i' = \|\mathsf{c}\|$ for Circular; otherwise for non circular	Circ Noncirc
Interface property	Definitions	API Parameters
Variability	$\|\mathsf{c}\| = size$ where $size$ is a constant, for Fixed; Otherwise for Varieble	Fixed Variable
Iterable	∃`iterator` to iterate elements in the collection with forward direction ; otherwise for Non-iterable ∃`reverse iterator` to iterate elements in a reverse order for an iterable collection	Iterable Noniter Bi_Iterable
Accessibility	$\exists at(i_x) = x$, where $i_x \forall i \in [0, size - 1]$ and i_x is the corresponding index, for randomly access; $\exists at(key)$ and $hash(), at(key) = x = hash(key)$ for key-value access; $\exists next(cur) = cur + 1$ where cur is current postion, for an Iterable & Ordered collection	Rnd Hashable Seq
Splitability	$splitAt(pos((x_1), pos(x_2), ...pos(x_n)), x_n \in \mathsf{c}$ $splitAt(x_1, x_2, ...x_n), x_n \in \mathsf{c}$	Nonsplit,SplitN, Splitable
UnionFind	$union()$ and $find()$ operations on disjoint sets	UnionFind
Hybrid property	Definitions	API Parameters
Order	$\exists pos(x) < pos(y) \forall x, y \in \mathsf{c}$ for Ordered; otherwise for Unordered $pop(x) \leq pop(y) \Rightarrow put(y) \leq put(x)$ for Last-in-first-out; $pop(x) \leq pop(y) \Rightarrow put(x) \leq put(y)$ for First-in-first-out; $pop(x) \leq pop(y) \Rightarrow comp(x) < comp(y)$ for First-in-best-out $x < y \Rightarrow pos(x) < pos(y) \forall x, y \in \mathsf{c}$ for Order by value	Ordered Unordered LIFO FIFO FO OrderByValue

2.1 Semantic Properites

Semantic properties affect the behaviour of the methods with which collections are accessed or modified. For our minimal prototype library we have implemented the following semantic properties:
Uniqueness A collection can contain duplicated elements if there are two or more equal elements, otherwise it is unique (e.g., a set contains unique values). In Table 1 , function `val()` returns the value of an element.
Circularity A collection is circular if the last and first positions are connected, otherwise it is non-circular.

2.2 Interface Properties

Interface properties specify certain functionality, usually in form of access methods to be provided by the collection. In this work we consider the following properties of this kind:
Variability A collection is variable if its size can be changed after construction, e.g. through a function `insert`; otherwise it is fixed, having an invariant size during its life cycle.
Iterable An iterable collection can be iterated (i.e., accessing its elements one after the other) through an iterator.
Accessibility This property specifies how elements of a collection can be retrieved. For random access a `[]` operator with which the user can *get* and *set* (if **const** keyword is not specified) is provided. For sequential access through an iterator we provide a *Seq* property, and for associate element accesses, we provide a *hashable* property by which an element is accessed via `[]` operator on an associative key.
Splitability If an ordered and non-circular collection can be split at some position or index to other parts, then it is a splitable collection. This is encapsulated in the behaviour of function `splitAt()`.
UnionFind This property provides *union* and *find* operations, which operate on set-like collections and provide a collection abstraction to disjoint sets.

2.3 Hybrid Properties

Certain properties are of hybrid nature, i.e. they both specify access methods and also change the way operations behave semantically. An example of such a hybrid property is *order*:
Order This property is an interface property as LIFO or FIFO order provide pop/push and queue/deque operation pairs, respectively. However, order is simultaneously a semantic property as the specified order influences the behaviour of the iterator *next* function. Iterator based traversals of the collection will yield different traversal orders depending on the specified order (insertion/extraction order, ordered by value, LIFO, FIFO, unordered).

With Collection Skeletons, it is straightforward to define a data collection only with the combinations of the desired properties in a declaritive way. To integrate the Collection Skeletons with modern C++ programming practice and make the learning curve for end users as smooth as possible, we implemented the API of the Collection Skeletons with C++ template meta programming, where a collection can be defined by a series of template parameters.

3 Library Design Principles

Our prototype Collection Skeletons library consists of two essential components: (i) A programming API that receives properties as declarative abstractions, and that is implemented based on C++ template meta programming, acting as the front end of the programming model; (ii) A multi-staged pattern matching based mapping mechanism between the declarative abstractions and the concrete internal data structures, which are currently based on C++ STL, Boost collections and other data structure libraries.

3.1 Programming API

We use C++ template metaprogramming for the implementation of our Collection Skeletons library. However, unlike the C++ STL, which also utilises template metaprogramming, we do not expose this to the application programmer. We have designed the API so that it supports the programmer in filling the template class with properties as type parameters. Thus, the user only needs basic knowledge on C++ programming to use our API.

Figure 3 provides an overview of the Collection interface as provided by our prototype library. Here we denote the property parameters enclosed in the angle brackets as the *property list*. In this API, the first type parameter `T` is the type of the elementary data to be stored by the data collection. Following `T`, P_1,P_2 to P_n are different properties of the desired data collection. These parameters of properties can found in the column *API Parameters* from Table 1.

A special case exists when the parameter *Fixed* is provided, i.e. the requested collection is *Fixed*, an unsigned integer constant must be provided through a non-type of namespace `size` as the fixed size of the collection.

Variadic type parameters `(F...)` are optional type parameters that may need to be applied when declaring a data collection, e.g., an OrderByValue data collection where the user requests a collection ordered by a value as specified by a user-defined comparison function. Such a user-defined comparison function can be passed as input through the variadic type parameters `F...`.

Table 2 summarises interface properties and their corresponding member functions. For convenience, we have defined a set of default methods available to all collections, which are described in Table 3.

Besides directly declaring the data collection with property list, users can also introduce a type alias to the property-based representation to simplify further usage such as,

```
using Collection<T,P₁,P₂,...Pₙ> = C1
```

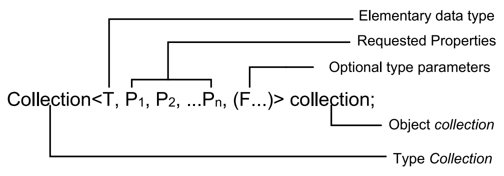


Figure 3. Overview of the Collection interface

where `C1` is a type alias. With the type alias, the user can declare a data collection in the following program without repeatedly writing down the same complex Collection type. This would e.g. allow the user to introduce a *set* alias for a collection with the properties of a set.

For ease of use, we assume in our prototype library that all the data collections are by default Duplicated, Non-Circular, Iterable, and Ordered unless explicitly specified otherwise.

3.2 Multi-Staged Pattern Matching and Mapping Algorithm

Selected combinations of properties need to be checked for consistency, and mapped onto available concrete data structures for implementation. This is the purpose of a multi-staged pattern matching algorithm described in this section.

With a list of properties as an input according to Figure 3, our library employs multi-staged pattern matching on the property list and eventually selects a concrete data structure providing those properties. If none of the available implementation data structures satisfy the requested properties the compilation will be interrupted with an error message. When a combination of property parameters can be satisfied by at least one concrete implementation, we call this combination of properties *eligible* and the implementation data structures become *eligible* candidates. An overview of this process is shown in Figure 4.

Conceptually, this process works akin to a database query – the user-provided input property list is transformed into a *query*, and our library performs a lookup over available implementation data structures and their provided properties through pattern matching. If a query can be matched, i.e. the input property list is eligible, we return a concrete data structure candidate and resume compilation; otherwise, if the requested properties are found to be internally inconsistent or no suitable implementation data structure that satisfies all of the requested properties can be found, compilation will be interrupted with an error message.

More detail of the pattern matching process is provided in algorithm 1. In fact, rather than returning suitable implementation data structures directly, we introduce another level of indirection through a dispatcher. This is for situations when more than one suitable implementation data structure can be found. Implementation candidates are wrapped in an

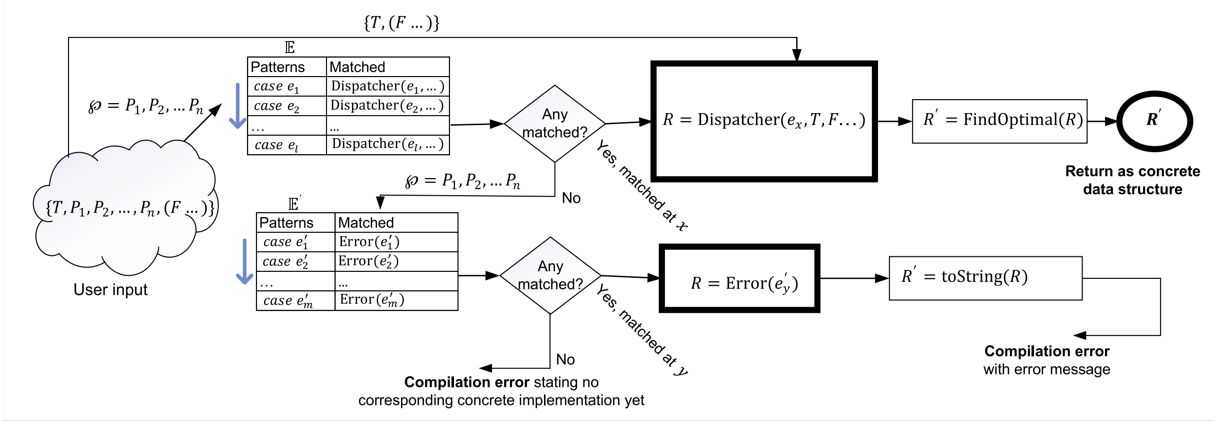

$\{T, (F \dots)\}$
$\mathbb{E}$
Patterns
Matched
$case\ e_1$
$\text{Dispatcher}(e_1, \dots)$
$case\ e_2$
$\text{Dispatcher}(e_2, \dots)$
$case\ e_l$
$\text{Dispatcher}(e_l, \dots)$
$\wp = P_1, P_2, \dots P_n$
Any matched?
$R = \text{Dispatcher}(e_x, T, F \dots)$
$R' = \text{FindOptimal}(R)$
R'
Return as concrete data structure
Yes, matched at x
No
$\{T, P_1, P_2, \dots, P_n, (F \dots)\}$
User input
$\mathbb{E}'$
Patterns
Matched
$case\ e_1'$
$\text{Error}(e_1')$
$case\ e_2'$
$\text{Error}(e_2')$
$case\ e_m'$
$\text{Error}(e_m')$
Any matched?
$R = \text{Error}(e_y')$
$R' = \text{toString}(R)$
Yes, matched at y
No
Compilation error stating no corresponding concrete implementation yet
Compilation error with error message

Figure 4. Resolving Properties to Concrete Data Structure Implementations through a Mutli-Staged Pattern Matching Algorithm

Table 2. Interface properties and their corresponding member functions

Properties		Guaranteed functions	Explanations
Variability	Variable	`void insert(Iter iter, T elem)`	Insert `elem` at position `iter` of collection Fixed or `const` collection can not be inserted elements after initialisation. For Collections with different interface properties, `insert` will be extended.

Iterable	Iterable	`Iter begin()`
		`Iter end()`
		`Iter next(Iter iter)`
			`begin()` returns an iterator to the beginning of an iterable collection
			`end()` returns an iterator to the end of an iterable collection
			`next()` returns the (immediately)next iterator/element of the current iterator(iter)

	Bi-Iterable	Apart from `begin()`, `end()`, and `next()`
		`Iter prev(Iter iter)`
		`Iter rbegin()`
		`Iter rend()`
			`prev()` returns the (immediately)previous iterator/element of the current iterator(iter)
			`rbegin()` returns a reverse iterator to the beginning of an iterable collection
			`rend()` returns a reverse iterator to the end of an iterable collection

Accessibility	Random	`Iter operator[](size_t i)`	Access the element through an index
	Hashable	`Iter operator[](Key key)`	Access the hashed result through a key
	Seq	`Iter next(Iter iter)`	Access the next element based on current element in an *Iterable & Ordered* Collection
Order	FIFO LIFO FO	Iter extract() void put(T elem)	First put one gets extracted first Last put one gets extracted first Extracting order defined by a function
UnionFind	UnionFind	`void union(C<T> set)` C<T> find(T elem)	Perform union operations on disjoint sets. Find if `elem` is in the disjoint sets and return the set that contains it

Splitability	Splitable	`tuple<Iter> splitAt(Iter iter)`
		`tuple<Iter> splitAt(T elem)`
			Split a collection at `iter`
			Split a collection at `elem`

Table 3. Default member functions

Default methods	Explanations
`size_t size()`	Return the size of a collection
`bool isEmpty()`	Check if a collection is empty, `true` for empty, otherwise not empty
Collection()	Default Constructor
Collection(size_t size)	Constructor by size
Collection(const Collection& c)	Copy constructor

Intermediate Class, which ultimately returns one of the eligible implementations. Incidentally, this intermediate class concept also provides us with a convenient way of providing extensibility to our Collection Skeletons framework as new data structure implementations can be added conveniently without change of the programming model or any user application code.

For example, a might user request a sequential accessible collection as follows:

```
Collection<int, Seq> c
```

Both a double-linked and single-linked list meet the requirement. In fact, in our prototype library, there are three concrete data structure candidates wrapped from `std::list`, `std::forward_list` and `boost:slist` available, which satisfy the requested property. The corresponding structure of the relevant intermediate class is shown in Figure 5. This class acts as a type alias for the three template classes associated with Macros. With predefined macro configurations passed through the function `FindOptimal`, a single concrete data structure will be selected and returned.

In our current prototype we resolve the selection of the implementation data structure through Macros and C++ template meta programming at compile time, but this step could be automated and even performed adaptively at runtime, e.g. using the CollectionSwitch framework [10].

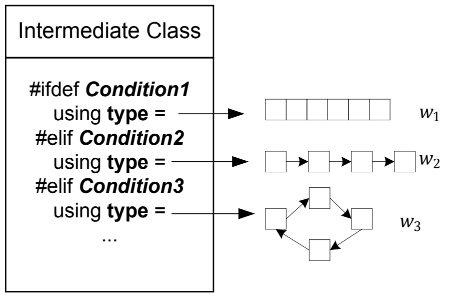


Figure 5. Example overview of an intermediate class

Algorithm 1: Property-based data collection deduction

```
Input  : Properties $\mathscr{P}$=$P_1$,$P_2$...$P_n$,
         Elementary data type $T$,
         Other template arguments, $(F...)$
Output : The implementation or error information
1  for $e \longleftarrow \mathbb{E}$ do
2      if $\mathscr{P} \equiv e$ then
3          $R \longleftarrow$ Dispatcher($e, T, F...$)
4          $R' \longleftarrow$ FindOptimal($R$)
5          return $R'$
6  for $e' \longleftarrow \mathbb{E}'$ do
7      if $\mathscr{P} \equiv e'$ then
8          $R \longleftarrow$ Error ($e'$)
9          return compilation error toString($R$)
10 return Combination toString($\mathscr{P}$) not yet
       supported
```

3.2.1 Detailed Discussion of the Matching Algorithm. The algorithm accepts the declarative properties as well as the elementary data type and other optional types parameters as input. In algorithm 1, $\mathscr{P}$ is the set of properties **parameters** declared; T is the elementary data type to be stored in the data collection, and F... are optional type parameters that maybe requested by the user. From line 1 to line 5, pattern matching of the eligible combinations of property parameters is performed, where $\mathbb{E}$ is the set of all eligible patterns of eligible combinations. Each time a pattern e is checked against $\mathscr{P}$. This algorithm then decides whether a pattern matches, i.e. a case has been found, through

$$\forall x : x \in e \iff x \in \mathscr{P} \implies \mathscr{P} \equiv e$$

After a case has been found, function `Dispatcher` in line 3 returns an **intermediate class** R based on the combination e and type parameters T, and F(if any). R stores all the information regarding the properties, types, type traits, and Macros that can be used by function `FindOptimal` to help deduct the final concrete data structure. R stores templates for many concrete data structures, and new data structures can be integrated to it without much work. Thus, the collection skeletons are flexible as one exact declaration can map to different concrete data structures, and with function `FindOptimal` we can get the optimal underlying data structure. If, however, no match has been found in line 1 to line 5, then the algorithm 1 will go to the next stage – pattern matching for erroneous declarations.

Expected common user errors such as putting a pair of *twofold symmetrical dichotomy* properties into the same property list, make that declarations ineligible. We have encoded the error combinations in $\mathbb{E}'$ and if a match is found, it means the input declaration cannot be resolved to a concrete data structure, and then a function `Error` generates error information based on the pattern and stores the error information in an intermediate class for error usage, denoted as R again. After that, a compilation error will be triggered with some human-readable information generated by function `toString` with R. After pattern matching at the error stage, if neither a concrete data structure is returned nor compilation error triggered during line 6 to line 9, that means the combination of properties is eligible but no concrete data structure could be found. For example, in our prototype we have not yet support for a collection that is of fixed size, ordered, and hashable. A compilation error will be triggered with a message stating that the input properties are not supported.

3.3 Rules of Properties and API Design

We have deliberately kept the programming API simple and user friendly, however, some rules are applied to prevent non-deterministic behaviour at code generation time, or more specifically during the stage of mapping to concrete data structures.

3.3.1 The Property List Is Order-Free.

Sequences of property parameters in the property list are order-free, for example

Collection<T, P_1, P_2, P_3>

works exactly as

Collection<T, P_2, P_3, P_1>

where P_1,P_2, and P_3 are different properties that together with T form an effective combination. Any permutation on properties P_1, P_2, and P_3, will eventually resolve to the same result, i.e. the same concrete data structure or the same error message.

3.3.2 Mutually Exclusive Properties Cannot Co-Exist in a Property List.

Some properties are mutually exclusive, e.g. twofold symmetrical dichotomy properties, where a property list cannot have both at the same time. For example, a collection cannot be *Ordered* and *Unordered* at the same time, nor can a collection allow *duplicates* and *unique* elements at the same time. For example, a collection declared as

```
Collection<int, Dup, Unique> c
```

is not permitted. As a result, if mutually exclusive properties are declared in the property list, e.g. the above ineligible declaration, compilation will terminate with an error message. These mutually exclusive cases are encoded in the error patterns as stated in line 6 to line 9 of algorithm 1.

3.3.3 No Guarantee on Algebraic Operations for Properties.

Users might expect to perform algebraic operations on properties or property lists. However, our library does not provide this facility at this stage. It is the user's responsibility to ensure consistent use of collections when extending functions or declaring their own functions that operate on the collections. For example, given two collections,

Collection<T, P_1, P_2, P_3, P_4> c1;

and

Collection<T_1, P_5, P_6, F> c2;

where both declarations are eligible, T and T_1 are elementary data types, F is an optional template argument. A user might want to define a `merge` function that merges the two collections `c1` and `c2`. At this stage we do not regulate the resulting type of the merged collection if the original collections comprise different property lists. Eventually, we feel the semantic effects of such operations on distinct collection types should be resolved by the user who must specify the intended behaviour of such an operation.

3.4 Implementation of the Pattern Matching Algorithm

As is shown in subsection 3.2, after receiving the declarative representation through the front end, the pattern matching algorithm operates on the properties to determine the resulting concrete data structure. We also implemented the pattern matching algorithm based on C++ template and type traits, thus no modification has been done on the compiler itself, making the prototype library more flexible.

```
template<typename T, typename... Ts>
struct contains: std::disjunction<std::
    is_same<T, Ts>...> {};
```

Listing 4. Check if type is contained in variadic type list

Listing 4 presents a template class that operates on type parameters to decide if a given type T is contained in the variadic type list F.... Thus, a pattern matching on the type parameters can be implemented based on the helper functions in Listing 4.

```
/*type selects*/
template<typename... Args >
using selects = typename std::disjunction<
    Args...>::type;

/*type when to decide the conditions*/
template<bool V, typename T>
struct when {
    static constexpr bool value = V;
    using type = T;
};
```

Listing 5. Compile time structures for pattern matching

```
        selects<
  when<Condition 1, type 1>,
  when<Condition 2, type 2>,
  when ...
              >
```

Figure 6. Grammar of pattern matching implementation

Figure 6 presents the grammar of the pattern matching implementation. In Listing 5, `selects` and `when` are the implementation of the basic pattern matching grammar - *when* the case matches, the corresponding class, i.e. the intermediate class will be *selected*.

```
struct CollectionTypeDispatcher{
    using type = selects<
        when<contains<Rnd, F...>::value && !
            contains<Fixed, F...>::value,
            typename pRnd<T>::type >,
        when<contains<Seq, F...>::value && !
            contains<Rnd, F...>::value,
            typename pSeq<T>::type >,
        ...
        ...
    >
}
```

Listing 6. Pattern Matching

Listing 6 shows some examples of the patterns and their corresponding intermediate classes. In Listing 6, for example, the first when expression, if the condition is true, i.e. Variadic types pack contains *Rnd* and do not contains *Fixed*, then an intermediate class pRnd will be returned as the result of the pattern matching.

```
template<typename T>
struct pRnd{
#ifdef WRAPPERSSTL
    using type = wvector;
#else
    ...
#endif
};
```

Listing 7. Intermediate Class

Listing 7 shows an example of the intermediate class, which is similar to Figure 5 mentioned before. Similarly, we implement the pattern matching for the incorrect combinations of properties in the same way except that error messages are enclosed in their corresponding intermediate classes. For the eligible combinations not yet supported by our library, we exploit static_assert to interrupt the compilation and return a message to the user as described in algorithm 1.

The Collection Skeleton library is fully implemented with C++ and operates at the compile stage, meaning it can be flexible and efficient.

We draw on the C++ STL, Boost and other data structure libraries for our pool of concrete data structure implementation. Through the provided wrapping mechanism it is trivial to extend our library with additional implementations, whilst introducing only minimal runtime overhead.

4 Evaluation

We evaluate our Collection Skeletons library against benchmark suites including Olden[4] and other open-source projects. We have manually rewritten these legacy applications to make use of our Collection Skeletons in order enable us to test our Collection Skeletons in real-world scenarios. For this we measure any performance overhead introduced by our novel abstractions in a set of before/after experiments.

For the purpose of this evaluation we have manually rewritten the legacy benchmarks written in C, but have replaced the low-level user-defined data structures and their access functions with their equivalent collections from our framework. No other code rewriting has been performed and the same input data have been used to facilitate a like-for-like comparison. Details of the benchmarks can be found in Table 4, where we have manually identified the original data structures from the benchmarks and their properties from the problem domain. With the extracted properties, we have replaced the original data structures with the declarative counterparts from our library. Column *Replaced* are concrete data structures before wrapped and obtained by manually checking the declaration and the pattern matching rules. In column *Replaced(Underneath)*, list, set, vector, map, queue, stack, and priority_queue are from the C++ STL, circular list and disjoint_set are from the Boost library.

We use Clang 12 and Clang++ 12 for compilation of the benchmarks, along with the "-O2" compiler flags. Besides, we enable C++ 17 compiling by setting "-std=c++17" for Clang++ 12. Experiments have been run on an Intel NUC 10(Intel Desktop), and 72-core Intel Gold 6154(Intel Server), and a 4-core Ampere Altra platform(Arm Server), respectively.

4.1 Experimental Results

4.1.1 Overall Performance.

Figure 7 presents the results of the speedup for the 17 benchmarks. From Figure 7 we can see that most of the replaced benchmarks have similar or better computational performance than the baseline, meaning little-to-no overhead has been introduced by our Collection Skeletons. Some of the replaced benchmarks have much better performance than the baseline ones, suggesting that for some benchmarks, by replacing the data structures of the original benchmarks, the performance can be greatly improved. For some benchmarks, the speedup is not substantial, e.g. the replaced ising benchmark only ran 4% faster than the baseline. Since ising benchmark is a small benchmark and does not consist of complex structures, the speedup is still important to be recognised. The average speedup of the 17 benchmarks across the three architectures is between 2.57-2.93.

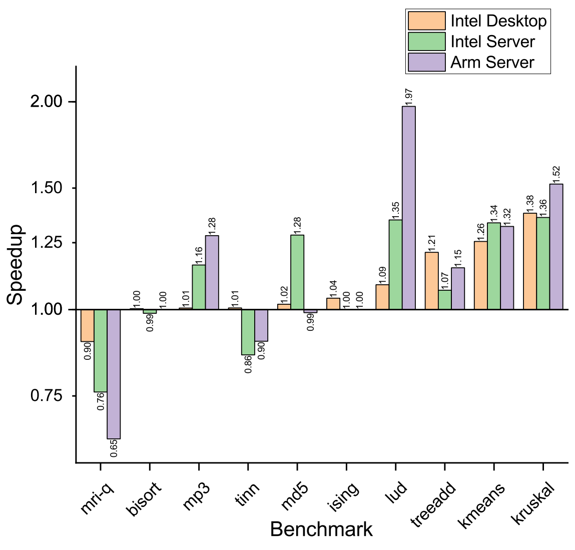
Although there are differences in speedup on the three architectures, e.g., for joseph benchmark, the speedup on the three architectures is 11.20, 16.37, and 10.36, the overall results show that the replaced benchmark have similar or better computational performance for almost every benchmark on the three architectures, which supports that the library has introduced little-to-no overhead and can even increase performance.

4.1.2 Implementation Flexibility.

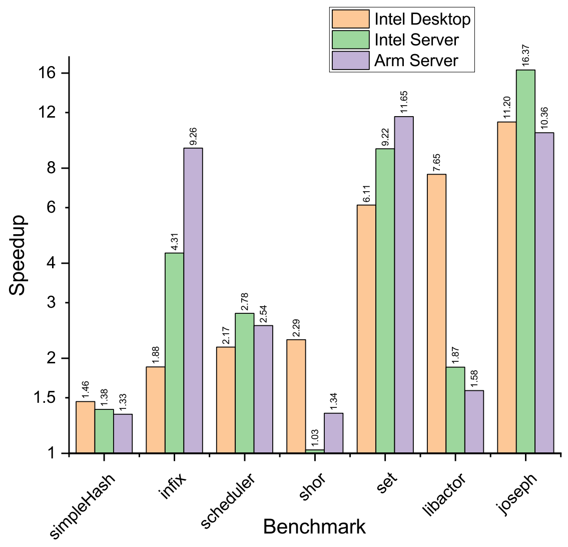
As mentioned in section 3 a single collection with a property list can lead to more than one concrete data structure as candidates. Here we have selected those benchmarks where this is the case and we have evaluated the computational performance for each of the feasible concrete data structures. We have found that benchmark treeadd, bisort, ising, set, libactor, tinn, shor, simpleHash, mp3, lud, kmeans and mri-q can be implemented using different concrete data structures from our library. As the mapping and deduction process is transparent to the user, there is no need to modify any user code, but we only need to modify some compile-time macros to control the

Table 4. List of the benchmarks, their data structures, extracted properties and replaced implementations

Benchmark	Source	Original Data structures	Extracted properties from problem domain	Replaced(Underneath)
treeadd	Olden[4]	balanced binary tree	Split2	tree2
ising	Github[7]	linked list	Seq	list
set	Rosseta[8]	linked list	Unique	set
libactor	Github[13]	linked list	Seq	list
tinn	Github[18]	array	Rnd	vector
shor	Github[23]	linked list array	Rnd	vector
simpleHash	Github[16]	linked list (of linked lists)	Rnd,Seq	vector(of lists)
mp3	Github[27]	linked list	Unique	set
scheduler	Github[19]	min heap	FO, OrderByValue	priority queue
md5	Github[14]	hashmap	OrderByValue, Hashable,Unordered	map
bisort	Olden[4]	binary tree	Split2	tree2
lud	Rodinia[5]	array	Rnd	vector
kmeans	Rodinia[5]	array	Rnd	vector
mri-q	Parboil[25]	array	Rnd	vector
joseph	Github[3]	circular linked list	Circ, Seq	circular list
infix	Github[12]	queue stack	Noniter,FIFO Noniter,LIFO	queue stack
kruskal	Geeksforgeeks[1]	map and array	UnionFind	disjoint_set



(a)



(b)

Figure 7. Speedup of the benchmarks on three platforms

behaviour of the function `FindOptimal` operating on intermediate classes, to *swap out* the concrete implementations. The experiments have been performed on the three platforms following the same method and the experimental results are shown in Table 5.

From Table 5, we can conclude that for different target platforms, the optimal underneath implementations can be different, e.g. slist from Boost library is the optimal data structure on Intel the desktop while list from STL is the optimal data structure on the Intel server. Furthermore, for some benchmarks, e.g. treeadd and bisort, the default chosen collection is a binary tree implemented through pointers, which is the de facto standard way; however, array-based binary tree has better speedup according to Table 5.

4.1.3 Performance Influencing Factors.

The size of the data collection, i.e. the number of elements stored in the collection, can have an impact on its performance and different

Table 5. Optimal implementations for selected benchmarks

Benchmark	Intel Desktop		Intel Server		Arm Server	
	Optimal	Speedup	Optimal	Speedup	Optimal	Speedup
treeadd	array_tree	1.61	array_tree	1.09	array_tree	6.37
bisort	array_tree	1.21	array_tree	1.33	array_tree	1.54
ising	slist	1.08	list	1.00	list	1.00
set	unordered_set	6.09	unordered_set	9.22	unordered_set	11.65
libactor	list	7.65	list	1.87	list	1.58
tinn	vector	1.01	vector	0.88	vector	0.9
shor	vector	2.29	vector	1.23	vector	1.34
simpleHash	forward_list	1.61	forward_list	1.44	forward_list	1.36
mp3	flat_set	1.01	set	1.16	set	1.28
lud	vector	1.09	vector	1.34	vector	1.97
kmeans	vector	1.26	vector	1.34	vector	1.32
mri-q	vector	0.90	vector	0.76	vector	0.65

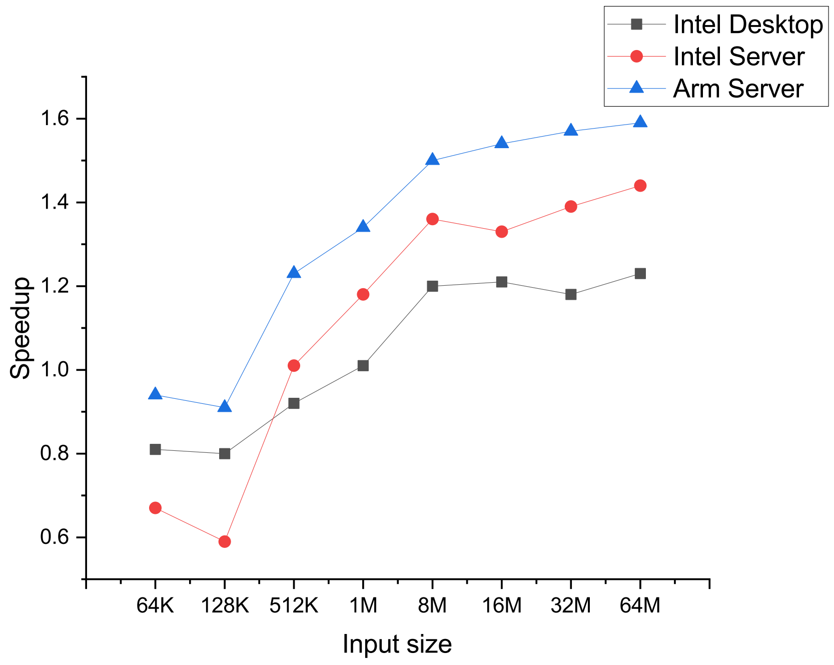


Figure 8. Speedup by array-based binary tree for bisort regarding the input size

architectures may offer different performance trade-offs. We further explore this for the *bisort* benchmark in Figure 8. The performance for our three target platforms is shown for up to 16M elements, where we treat the original pointer-based implementation(`tree2`) as a baseline to evaluate the relative performance of an array-based binary tree(`array_tree2`). For up to around 512k elements we observe a slowdown of the array-based binary tree over the original pointer based implementation, but for larger collections the array-based implementations outperform the original implementation by up to 60%. While we exercise in our experiments manual control over the final implementation through passing different Macros to function `FindOptimal`, an automated technique such as "CollectionSwitch" [10] could be employed to automate this step.

5 Related Work

Abstract data types provide a mathematical model of data types and are defined through semantics of data access functions. In practice, there exist greatly different implementations of the ADTs as there are no universal definitions of even the most commonly used ADTs.

Transparently replacing and dynamically switching data collections has been proposed in several papers, e.g. [10, 11, 15, 28]. However, none of these works expose collection properties to the user, but use quantitative runtime data to select an optimal implementation for a specific runtime scenario. These works are typically Java based.

In [20], collection libraries from 14 languages are reviewed. In their work, the authors discuss properties of collections, which has partly inspired our work.

In [2], a framework that uses Concern-Oriented Reuse (CORE) to capture many different kinds of associations, their properties, behaviour and various implementation solutions within a reusable artifact has been proposed. Their work has focused on software reuse, which also inspired our work regarding portability.

C++ "concepts" have been introduced in C++ 20, specifying the requirements on template arguments which can also be used to specify the properties of a template class [26]. However, in contrast with our collection skeletons, C++ concepts require the end-user to have a deep understanding of C++ meta programming, which is not the case for our novel abstraction library.

To the best of our knowledge, no previous work has proposed to classify properties of data collections based on their semantics and interface functions, and attempted to develop a truly declarative approach of providing the user with data collections.

6 Summary & Conclusions

We have introduced a declarative approach to specifying data collections by exposing fundamental collection properties to the programmer. We show that our property based approach to collections does not introduce performance overhead, but instead opens up the opportunity for performance improvements gained through greater implementation flexibility, which is hidden from the application programmer. Benchmark applications rewritten to make use of our novel Collection Skeletons show favourable performance characteristics across different target platforms.

In our future work we will explore the interaction of our Collection Skeletons and Algorithmic Skeletons, opening up further performance benefits from parallelisation. We will also explore avenues to make data structure selection more adaptive to the target machine and application context, e.g. through the use of machine learning methods for optimal selection of an optimal implementation data structure at runtime.

Acknowledgements

The authors would like to thank the anonymous reviewers for their constructive suggestions and comments. They would also like to thank Chris Vasiladiotis for his help with the revision. This work was supported by the grant from Huawei-Edinburgh Joint Lab.

References

[1] Chirag Agrawal. 2021. kruskal disjoint. https://www.geeksforgeeks.org/kruskals-minimum-spanning-tree-using-stl-in-c/.
[2] Céline Bensoussan, Matthias Schöttle, and Jörg Kienzle. 2016. Associations in MDE: a concern-oriented, reusable solution. In *European Conference on Modelling Foundations and Applications*. Springer, 121–137. https://doi.org/10.1007/978-3-319-42061-5_8
[3] Manish Bhojasia. 2013. joseph. https:www.sanfoundry.com/c-program-solve-josephus-problem-using-linked-list.
[4] Martin Christopher Carlisle. 1996. *Olden: parallelizing programs with dynamic data structures on distributed-memory machines.* Princeton University.
[5] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE international symposium on workload characterization (IISWC)*. Ieee, 44–54. https://doi.org/10.1109/iiswc.2009.5306797
[6] Lin Chen, Di Wu, Wanwangying Ma, Yuming Zhou, Baowen Xu, and Hareton Leung. 2020. How C++ templates are used for generic programming: an empirical study on 50 open source systems. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 29, 1 (2020), 1–49. https://doi.org/10.1145/3356579
[7] Chris. 2020. ising benchmark. https://github.com/compor/mischung-suite/blob/master/programs/ising/data/original_source/ising.c.
[8] Rosseta Code Contributors. 2022. Rosetta Code — Rosetta Code,. https://rosettacode.org/wiki/Rosetta_Code [Online].
[9] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2001. *Introduction to Algorithms* (2nd ed.). The MIT Press.
[10] Diego Costa and Artur Andrzejak. 2018. Collectionswitch: A framework for efficient and dynamic collection selection. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*. 16–26. https://doi.org/10.1145/3168825
[11] Diego Costa, Artur Andrzejak, Janos Seboek, and David Lo. 2017. Empirical study of usage and performance of java collections. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*. 389–400. https://doi.org/10.1145/3030207.3030221
[12] Diego. 2021. infix to postfix. https://github.com/dlb04/infix-to-postfix.
[13] Jim Fisher. 2011. libactor. https://github.com/airplug/libactor.
[14] Damian Jarek. 2015. libactor. https://github.com/djarek/md5lamacz.
[15] Changhee Jung, Silvius Rus, Brian P Railing, Nathan Clark, and Santosh Pande. 2011. Brainy: Effective selection of data structures. *ACM SIGPLAN Notices* 46, 6 (2011), 86–97. https://doi.org/10.1145/2345156.1993509
[16] Chris Lattner and Lauro Venancio. 2021. simple hash benchmark. https://github.com/llvm-mirror/test-suite/blob/master/SingleSource/Benchmarks/Shootout/hash.c.
[17] Barbara Liskov and Stephen Zilles. 1974. Programming with Abstract Data Types. In *Proceedings of the ACM SIGPLAN Symposium on Very High Level Languages* (Santa Monica, California, USA). Association for Computing Machinery, New York, NY, USA, 50–59. https://doi.org/10.1145/800233.807045
[18] Gustav Louw. 2020. tinn. https://github.com/glouw/tinn.
[19] Jason Marcell. 2009. scheduler. https://github.com/jasmarc/scheduler.
[20] Stefan Marr and Benoit Daloze. 2018. Few versatile vs. many specialized collections: how to design a collection library for exploratory programming?. In *Conference Companion of the 2nd International Conference on Art, Science, and Engineering of Programming*. 135–143. https://doi.org/10.1145/3191697.3214334
[21] Maurice Naftalin and Philip Wadler. 2006. *Java Generics and Collections.* O'Reilly.
[22] Martin Odersky and Lex Spoon. 2019. The Architecture of Scala Collections. https://docs.scala-lang.org/overviews/core/architecture-of-scala-collections.html
[23] rafa32. 2017. shor. https://github.com/rafa32/Quantum-Shor.

[24] Alexander Stepanov and Lee Meng. 1995. *The Standard Template Library*. Technical Report HPL-95-11(R.1). HP Laboratories.

[25] John A Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Daniel Liu, and Wen-mei W Hwu. 2012. Parboil: A revised benchmark suite for scientific and commercial throughput computing. *Center for Reliable and High-Performance Computing* 127 (2012), 27.

[26] Peter Thoman, Florian Tischler, Philip Salzmann, and Thomas Fahringer. 2022. The Celerity High-level API: C++ 20 for Accelerator Clusters. *International Journal of Parallel Programming* (2022), 1–19. https://doi.org/10.1007/s10766-022-00731-8

[27] Timmy Whelan. 2002. mp3. https://sourceforge.net/projects/mp3reorg/files/mp3reorg/.

[28] Guoqing Xu. 2013. Coco: Sound and adaptive replacement of java collections. In *European Conference on Object-Oriented Programming*. Springer, 1–26. https://doi.org/10.1007/978-3-642-39038-8_1