Few Versatile vs. Many Specialized Collections

How to design a collection library for exploratory programming?

Stefan Marr
School of Computing
University of Kent
Canterbury, United Kingdom
s.marr@kent.ac.uk

Benoit Daloze
Institute for System Software
Johannes Kepler University Linz
Linz, Austria
benoit.daloze@jku.at

ABSTRACT

While an integral part of all programming languages, the design of collection libraries is rarely studied. This work briefly reviews the collection libraries of 13 languages to identify possible design dimensions. Some languages have surprisingly few but versatile collections, while others have large libraries with many specialized collections. Based on the identified design dimensions, we argue that a small collection library with only a sequence, a map, and a set type are a suitable choice to facilitate exploratory programming. Such a design minimizes the number of decisions programmers have to make when dealing with collections, and it improves discoverability of collection operations. We further discuss techniques that make their implementation practical from a performance perspective. Based on these arguments, we conclude that languages which aim to support exploratory programming should strive for small and versatile collection libraries.

CCS CONCEPTS

• **Software and its engineering → Data types and structures**; *Abstract data types*; • **General and reference** → *Surveys and overviews*;

KEYWORDS

Collection Libraries, Design, Implementation, Exploratory Programming

PX/18, April 2018, Nice, France

ACM ISBN 123-4567-24-567/08/06…$15.00
https://doi.org/10.475/123$_4$

1 COLLECTIONS FOR EXPLORATORY PROGRAMMING

Collections or containers are part of all general purpose languages. They are data structures that abstract over the number of data items and allow operations over the contained items as a whole or individually.

While being fundamental to most languages, the design of collection libraries varies greatly. At one end of the spectrum, we have C, which has only arrays. At the other end, we have for instance Java, with a standard library that includes dozens of collection abstractions, and with external libraries to support different programming styles, application scenarios, or performance requirements.

To the best of our knowledge, there is no study on how the design of a collection library influences programmer productivity or the suitability of a language for exploratory programming.

The goal of this work is to identify the aspects that need to be considered for the design of collection libraries for exploratory and live programming. Thus, we reason about the design space for collections and how different design choices may influence the programming experience. Our analysis relates these design choices to the main programming task, performance, and the programming experience.

To get an overview of the design space, we discuss the high-level design of collection libraries for C, C++, Go, Java, C#, Scala, Haskell, Racket, Pharo Smalltalk, Ruby, JavaScript, Python, Dart, and Lua. From this overview, we extract design dimensions and argue for specific choices in the context of exploratory programming. We also review implementation techniques to support the design's practicality in terms of achieving the same performance as alternative choices.

While not exhaustive, we hope this analysis provides an insight into the complex design space of collection libraries and their relation to the programming experience itself. Our immediate goal was to better understand the tension between design choices for the design of a collection library for SOMns, a Newspeak implementation [5, 20]. Newspeak follows in many ways previous Smalltalk systems and as such has a large and specialized collection library. However,

based on the arguments presented here, we believe that a small and versatile library would be a more suitable design for a language that itself is quite suitable for exploratory programming.

2 LIVE AND EXPLORATORY PROGRAMMING

Before diving into the details of designing collection libraries, we discuss the terms live and exploratory programming to clarify our assumptions and the context of this analysis.

For the term *live programming*, we rely on Tanimoto's notion. Specifically, we aim to support programming with immediate updates and feedback from the programming environment. This might be either triggered by edits or presented as continuous event streams, which corresponds to liveness of level 3 or 4 [28]. While generally desirable, we forgo considering more advanced liveness levels that include predictive features, i.e., level 5 and 6 [29].

Based on these liveness level and on the work of others [7, 22, 27], we expect programming environments and languages to improve the programming experience by providing immediate feedback on the validity of programs and code execution. This feedback should be meant to reduce the friction that is inherent in the communication between humans and machines, which is constrained by artificial languages and machines that still neither perceive the context of a conversation nor learn from interaction. To work around these limitations, we expect machines to support the communication with feedback, limited forms of understanding, and a basic robustness to invalid communication.

Feedback should be provided in form of code completion, contextual documentation, syntax errors, highlighting of possibly invalid method calls or operations, style violations, and generally undesirable patterns. The analyses that enable such feedback need to be implementable in an efficient manner to give the illusion of instantaneous feedback. With limited forms of understanding, we refer essentially to analyses that allow programmers to be more implicit than explicit in their programming. One aspect of this is typing. Languages that use type inference or tool-based feedback seem to facilitate exploratory programming better than languages that require us to make such aspect explicit. The overall system should also be robust with respect to programming errors. Especially errors in unrelated program parts, or parts that are not executed should not inhibit our ability to interact with the system, explore, and extend it.

Generally, when exploring a problem and experimenting with solutions, we want to focus on the problem domain and prioritize the core aspects while deferring for instance overall application performance, handling of exceptional cases, or the completeness of a solution to a later point. Thus, the programming environment should not force us to make decisions upfront, for which we do not understand the problem well enough yet or for which we would predict highly inaccurately how an algorithm is used in production scenarios. Types or structural elements often help exploring the problem and organizing thoughts, but should not prevent or hinder us from exploring it.

Since collections provide bulk operations and are relevant for acceptable performance, run-time performance needs to be considered, even though it is a secondary concern.

3 COLLECTION DESIGN IN THE WILD

Collection libraries are as diverse as languages are different. There are many design dimensions to consider. Some dimensions are influenced by the underlying language, its basic abstractions, as well as what is considered desirable programming style, i.e., idiomatic code. Other dimensions include the set of offered collection types, implementation strategies (usually differing in space/time tradeoffs), or general properties such as synchronization, modifiability, ordering, sorting, and constraints in general. This section briefly reviews collection libraries of 13 languages to identify the provided collection types and high-level design differences.

C11. The C11 standard includes only a single collection type: arrays. In comparison to other languages, C arrays are merely raw memory. They neither include a length nor any form of safety checks, and thus represent the minimal possible abstraction. The GNU C Library[1] expands only minimally on arrays by providing hash table and tree-based operations on arrays for searching and sorted access.

C++17 and Boost 1.66. C++17 [19] expands on C by providing a template-based standard library (STL). It includes C arrays but also an STL version, which can do bounds checking. Additionally, it includes vectors, deques, singly and doubly linked lists, sets and maps in an ordered (tree-based) and unordered (hash-based) variant, as well as adapters to realize stack or queue semantics. Generally, the STL remains comparably lean, but already provides collections with different performance tradeoffs that enable programmers to choose based on their expected usage scenarios.

The Boost C++ libraries[2] expand on the STL by adding more specialized containers, which at least partially address C++-template-specific issues. The variations on the vector class address different performance tradeoffs. Overall Boost seems to focus on optimized collections for specific use cases.

[1] *The GNU C Library (glibc)*, GNU Project, https://www.gnu.org/software/libc/manual/

[2] *Boost C++ Libraries*, boost.org, http://www.boost.org/

Go 1.9. The Go language[3] comes with abstractions for arrays, lists, rings, maps, and heaps. The library includes only the basic abstractions needed. For instance, it does not contain a separate set type, perhaps because it can be easily modeled based on maps.

Java 8, Guava 24, and Eclipse Collections 9.1. In contrast, Java[4] has a large collection library. As basic abstraction, the language includes arrays, which can be of different primitive or object types. Its standard library provides implementations for a wide variety of collections including lists, maps, sets, queues, deques, priority queues, stacks, and iteration abstractions. Many of the collections come in different variations. For maps and sets, there are hash and tree-based implementations. Lists are implemented for instance as linked or array lists, which is then complemented with skip lists, or copy-on-write lists. Depending on the collection type, there are also variations that provide ordering guarantees, sorting, make them navigable, enforce read-only access, reference elements only weakly, or give guarantees for concurrent accesses. Concurrent accesses can be synchronized with a basic synchronization wrapper or by using a specific data structure that might have less synchronization overhead or allows for more parallelism. As a result, the standard library provides a large number of collection implementations. While some properties are provided by wrappers and allow for combination, many collections are separate implementations tuned for specific use cases.

Guava[5] extends the Java collection library with additional utility methods and adds multi-sets, multi-maps, bidirectional maps, tables, as well as range sets and maps. Furthermore, it adds immutable versions of the basic collection. The focus is on more efficient in-memory representations and improving performance for access operations compared to Java's read-only/unmodifiable collection wrappers.

The Eclipse Collections library[6] also extends the Java collection library. Similarly to Guava, it adds bidirectional maps, multi-maps, bags, and provide additional utilities. This includes rich support for iteration for instance to iterate lazily or in parallel. It also adds immutable collections. Additionally, it provides specialized collection types for primitive types to avoid boxing overhead of Java's collection types.

C# and .NET Framework 4.7. Similar to Java, C# and the .NET Framework[7] have arrays, lists, sets, maps, queues, and stacks. While the framework includes linked and array lists, it does not include explicit tree sets or maps. However, it also includes sorted collections, which might be implemented as trees, but the documentation refrains from specifying the performance of operations or naming an implementation technique. Similar to Java, C# provides read-only, i.e., unmodifiable collections, which are wrappers. Additionally, it provides immutable collections, which are implemented as persistent data structures, i.e., use structural sharing. For thread-safety, it offers synchronization wrappers as well as special-purpose concurrent collections with better concurrent behavior. One feature specific to C# and .NET is the support for query operations on collections with LINQ [23], which are an addition to the traditional iteration operations.

Scala 2.12. Scala's standard library[8] supports arrays, maps, sets, and sequences as basic abstractions. It got hash and tree-based versions for sets and maps as well as array-based or doubly linked lists to cover the main implementation strategies. It got special-purpose collections such as bit sets, list maps, integer maps, linked hash maps or sets, multi-maps, queues, stacks, vectors, and weak hash maps. The library offers mutable and immutable versions of most collections. The immutable collections are typically implemented as persistent data structures with structural sharing. Both types of collections are polymorphic on read operations, which makes them interchangeable for basic accesses. Additionally, the library provides concurrent, sorted, and parallel versions of some of the collections. The parallel collections focus on parallel execution of bulk operations.

Haskell. Haskell offers only immutable lists and tuples as builtin types. Other collections are provided as external packages. The *base* library, which is the standard library for the Glasgow Haskell Compiler (GHC) only contains channels, i.e., concurrent queues. The *containers* library, the de-facto standard collection library, provides sequences, maps, graphs, sets, trees, and bit queues. Some of these are provided as lazy or strict variants. Similarly, it provides special sets and maps for integers, presumably for performance reasons.

Racket 6.12. Racket[9] provides pairs and lists in the Scheme tradition. In addition, it provides vectors, boxes, hash tables, and sets. Most collections are available as immutable and mutable variants. To support concurrency, it provides channels. Buffered channels are synchronized queues.

Pharo 6. Pharo's collection library is based on the collection library of Smalltalk [11, 17]. It includes arrays, maps, sets, bags, stacks, matrices, and doubly linked lists. Some

[3] *The Go Programming Language*, Go Project, https://golang.org/
[4] *Java SE 8*, Oracle, https://docs.oracle.com/javase/8/
[5] *Guava: Google Core Libraries for Java*, Guava Project, https://github.com/google/guava/wiki/NewCollectionTypesExplained
[6] *Eclipse Collections*, Eclipse Foundation, https://www.eclipse.org/collections/
[7] *.NET API*, Microsoft, https://docs.microsoft.com/en-us/dotnet/api/?view=netframework-4.7.1
[8] *Scala Standard Library*, EPFL, http://www.scala-lang.org/api/2.12.4/
[9] *The Racket Reference*, Racket Project, https://docs.racket-lang.org/reference/

collections are available in variants that are ordered, sorted, or based on identity. It also offers arrays of primitive types that store raw values in memory. Small maps can use classes that are specifically optimized for this purpose. Similar to other languages, it includes weak collections as well as a number of queues for concurrent use.

Ruby 2.5. Ruby's collection library is comparably small. Its main collection types are arrays and maps. Instead of offering lists, deques, or stacks as separate collections, `Array` supports the corresponding operations. The `Hash` map always maintains insertion order, and can be instructed with `compare_by_identity` to use identity of keys. Ruby also provides variable size and bounded queues for multithreading and a range abstraction. Immutability of collections is supported by Ruby's mechanism to freeze objects. Sorting is not provided as part of separate collection types either. Instead `Hash` and `Array` have a `sort` method. Ruby's collections provide many internal iteration and transformation methods so that external iteration is rarely used.

JavaScript, ECMAScript 2016. JavaScript's collection library used to offer merely arrays and objects, which could be used as maps but only accept strings as keys. Since this was limiting, newer versions of the ECMAScript standard introduced maps, sets, and their weak counter parts. They also introduced typed arrays, i.e., views on raw memory that allow representing certain number types directly in memory.

Python 3.6. Python offers lists, immutable tuples, ranges, sets, and maps as basic collections. Since Python 3.6, `dict` maintains insertion order and makes `OrderedDict` redundant. To provide a single view on multiple maps, it offers `ChainMap`. To enable sets of sets, it includes a `frozenset`, which has the necessary support for obtaining a hash value. Priority queues and deques are also provided in standard modules. Similar to Racket, it provides synchronized queues for communication between threads. To represent numbers efficiently, it also supports typed arrays. A special case is here the `bytes` type, which is an immutable byte sequence that is used for instance for byte literals.

Dart 1.24. Dart includes lists, deques, sets, maps and queues in many variants. Sets and maps are available either using hash tables, linked hash tables, or splay trees. Similar to other dynamic languages, Dart also offers lists of primitive types for efficient representation in memory.

Lua 5.3. Lua[10] is designed as a lightweight language for embedding and scripting. The lightweightness is made explicit also in its approach to collections. The only abstraction it provides is a table, which is an associative array, but also used as basic abstraction for object-oriented programming. Consequently, tables are used for sequences and maps and have been optimized to fulfill both types of usages [18].

[10] *Lua 5.3*, Lua.org, https://www.lua.org/manual/5.3/

4 DESIGN DIMENSIONS FOR COLLECTION LIBRARIES

Based on the observations of section 3, this section distills the design dimensions for collection libraries.

4.1 Collection Types

The first dimension is the *collection types* to be included in the library. We abstract from general properties of collections (cf. section 4.3) and other implementation or representation choices (cf. section 4.5), which we consider as orthogonal concerns even so it is not always obviously beneficial.

The first category of collection types is *sequences*. Considering mutability, sizing, and representation as orthogonal concerns, this category includes arrays, lists, vectors, tuples, pairs and boxes. Generally, these collections provide the ability to store elements that are possibly repeating and operate on them perhaps by direct access or via iteration.

The second category is *sets*. Thus, collections that do not maintain repeated elements and only store a single occurrence based on some equivalence criterion.

The third category is *maps*, which maintain a mapping from keys to values. We put bags, i.e., multi-sets, in this category as they map keys to a number of occurrences. Languages such as Racket allow polymorphism between indexed sequences and maps (the key being the index).

The fourth category is *stacks and queues*. These collections store elements and restrict the access in a way that facilitates the efficient implementation of certain use cases. We include here deques, priority queues or heaps, and rings.

The fifth category is *composed collections*. These are collections that could be built by combining multiple collections. We include here for instance matrices, which are two-dimensional arrays, and tables, which are two-dimensional maps. Again, these two could be considered of the same class if indexing with integers is considered an associative access.

The six category is *ranged collections*, which are rare and appear for instance in form of range maps or sets.

4.2 Language Style

The second category with a major influence on the collection design is the language for which the library is designed.

The *language style* in terms of being procedural, object-oriented, or functional leads to different designs in shape and structure of collections as well as the provided sets of operations on them and the operation naming.

The language's stance on *typing* and its support for type parameters shapes libraries in various ways. In dynamically-typed languages, we see the need for explicit support of

collections for primitive types, e.g., to store numeric data efficiently. This includes sequences for numeric elements (e.g. `Int32Array` in ECMAScript) or primitive maps or sets (e.g. `IntMap in Scala`). Typed languages often use some form of generics, type parameters, or templates, which help to reduce a proliferation of collections for specific data types.

In addition to typing, the *reuse* mechanisms offered by a language have an important impact on library structure. For example, single inheritance can lead to designs with undesirable properties as seen in the Smalltalk-80 collection library [11]. While some of these issue can be worked around, other reuse mechanisms such as traits, might result in designs that have benefits [3].

In some case, languages evolved to facilitate desired collection library designs [8, 16]. In other cases, the library needed to evolve and use the available language mechanisms more effectively to improve maintainability and code reuse [24].

4.3 Properties

As noted in section 4.1, it is not always clear whether the properties of collections are orthogonal to the collection type, but they seem generally useful to be discussed separately.

Basic Properties. For some of the collection types, variants exist that have a *fixed or variable size*. For instance, sequences can be vectors with a variable size, or arrays with a fixed size. Similarly, queues might use a variable size implementation or a fixed sized to enforce a bound on the upper queue length.

For many collection types, there are variants that are *ordered or unordered*. Lists or other sequences commonly maintain insertion order, which can be equally beneficial for maps, sets, and other collection types.

In addition to ordering, collections can *sort elements*. Some collections such as tree-based collections maintain a sorted order at all times and provide operations taking advantage of this property. Other collections might provide an operation to sort elements on demand.

Mutability. Collections can differ in their mutability and be available in *mutable* and *immutable* versions. Some libraries additionally provide *read-only views* in form of wrappers.

Some immutable collections are implemented as *persistent* data structures, which use structural sharing to avoid high memory overhead when producing a new version of a large collection with only minimal changes.

Special-purpose variations include further *copy-on-write* data structures such as Java's `CopyOnWriteArrayList`, which can be used to isolate multiple entities and enable mutability.

Multi-Threading Support. To support multi-threaded applications, libraries can include various collection types. The simplest solution to ensure correctness is to provide *synchronization wrappers* as done by Java and C#. However, this approach is rarely efficient and often lacks support for performing multiple operations safely together. These problems are typically addressed by specialized *concurrent* collections, which are optimized for specific use cases, leading to a proliferation of collection types. For instance a queue for a single producer but multiple consumers can often be implemented more efficiently than a queue for multiple producers.

For efficient bulk operations, languages such as Scala provide *parallel* collections, which execute operations in parallel. Similar to sorting, where some collections do it intrinsically, parallel execution is often provided via operations external to the collection (cf. section 4.4).

Other Properties. Weak collections are commonly offered to build caches and other data structures that interact well with garbage collection by enabling reclamation of not otherwise referenced objects.

Languages such as Haskell offer the distinction between *lazy and strict* collections to provide control over when operations are executed. However, this distinction is not exclusive to lazy languages. In conjunction with iteration, other languages can also offer lazy execution.

Finally, we found sets and maps that use *identity* as main distinguishing criterion. Presumably, these collections are optimizations because Java, Smalltalk, and Dart would allow to construct maps that can be parameterized with appropriate operations for comparison and hashing by identity.

4.4 Operation Design

The design of collection operations depends on the language style (cf. section 4.2). The style influences naming, which operations are offered, how bulk operations are realized, or errors are handled. To give just one example, in Smalltalk it is common to pass closures as arguments to operations that might fail to provide the failure handling code directly. In other languages, return codes or exceptions indicate failure and are handled externally to the collection.

There are other previously mentioned aspects that influence the offered operations including immutability, persistence, sorting, parallel bulk operations, laziness, iteration in general, and possibly properties that are realized as some form of opt-in mechanism, which may include immutability and read-only views.

One important aspect of designing the interfaces of collections is the intended degree of polymorphism, i.e., whether collections should offer the same interfaces. To facilitate code reuse, one might want to design collections with different properties such as being mutable or immutable so that they can be used mostly interchangeably. One way of doing this is by using decorators or wrappers around collections that add the desired properties, e.g., read-only views or basic synchronization in Java. Other options to consider are whether

maps should be iterable in the same way as sequences, or whether sequences can be treated as maps.

Iteration or more general collection traversal is an important design point. The basic properties ordering and sorting need to be taken into account. Furthermore, iteration operations need to consider whether processing is to be done sequentially, in parallel, or lazily. Languages with slices or ranges can use these as interface to define how to iterate. Finally, other concepts such as internal vs external iteration, streams, and language-integrated query (LINQ) further influence the set of offered operations.

4.5 Algorithms, Data Structures, and Implementation Choices

The final set of design dimensions are implementation choices, which include the selection of a specific algorithm or data structure to realize a collection.

Literature describes a large variety of special-purpose data structures, especially for concurrent applications. However, there seem to be a few favorites that are recurring. For trees we saw for instance red-black trees, tries, and splay trees, for lists singly and doubly-linked ones, array-based, and skip lists, possibly with specializations for small or rarely changing lists. The concrete algorithms for hash tables are not usually specified in the documentation.

Another important design dimension mentioned before is how the various properties of collections are represented. Some libraries provide separate classes for each possible type and property, while others realize the desired properties with decorators, i.e., wrapper constructs.

5 COLLECTION USAGE

As seen in the previous section, collection libraries differ widely in the number of collection types they provide and the properties these collections can offer. Especially Java provides a multitude of different collection types and still, the wider community felt the need to provide and maintain libraries for additional collections. One important question to guide the design could therefore be: which collection types are widely used and are likely going to be needed for exploratory programming tasks?

To answer the question of which collection types are widely used, Costa et al. [13] studied a GitHub corpus of Java projects [1]. They found that most instantiation sites for collections create `ArrayList` objects. From all sites analyzed, 47% used the standard Java `ArrayList`. Overall, about 56% used some kind of list. Maps where used by about 28%, where the great majority uses Java's `HashMap`, which results in a total of about 23% of all allocations. About 15% all instantiation sites were for some set type. Again, the large majority was for Java's `HashSet` with about 10% of all allocations.

The only other study on collection usage we are aware of was included in work by Bergel et al. [2, sec. 9.2]. They observed that `OrderedCollection` (similar to `ArrayList`) and `Dictionary` (a map) are the most frequently used collections in some larger Pharo Smalltalk projects. While the study is less comprehensive than the Java one, it confirms the general trend. It also considers Smalltalk arrays and finds that they are used slightly more often than `Dictionary`.

6 A COLLECTION DESIGN FOR EXPLORATORY PROGRAMMING

As discussed in section 2, to facilitate exploratory programming a collection library needs to help programmers to 1) focus on the problem domain 2) avoid unnecessary decisions and 3) enable the environment to provide feedback in form of e.g., errors, hints, tooling, and documentation. To us, this means we either want to completely avoid a choice for or against a specific collection type or property, or postpone it as far as possible. At the same time, when we made certain choices, we want them to be reflected by the environment. Thus, independently of how these choices are expressed in code, the environment needs to recognize them to provide us maximal support. Furthermore, we consider performance in this context a secondary concern. While important for many applications, we assume it is rarely the primary concern for exploratory programming. Consequently, we prefer solutions that can provide perhaps 80% of a special purpose solution without exposing it to programmers.

The remainder of this section first discusses the design dimensions identified in section 4 to propose choices that fit our vision for an exploratory programming setting. Afterwards, we discuss the resulting overall design, possible criticism, and tradeoffs.

6.1 Identifying a Point in the Design Space

Types of Collections. The first and perhaps most important choice seems to be the selection of the desired set of collection types. In section 4.1, we identified six groups. However, considering the usage in actual applications (cf. section 5), only sequences, sets, and maps seem to be widely used. To avoid unnecessary decisions, we propose to use only a minimal set of collection types based on the most widely used ones, and to provide relevant functionality as part of these collections. Furthermore, we suggest to design collection properties in a way that one can easily opt into properties or opt out from them. However, this should be designed together with tool support to avoid losing relevant feedback from the environment. For instance, if a list is used as a stack, the environment should pick up on it, and adapt code completion accordingly.

It can be debated whether to include sets or not. Sets can be easily emulated with maps or with set operations on sequences. Since the use and semantics of sets is distinct enough from sequences and maps, and the various possible designs of emulating sets have drawbacks, we would include them directly, even though there are a number of languages that do not do so.

Similarly, one can argue that maps are merely lists of pairs or that all sequences should be associative arrays. We agree that it can be beneficial to threat them as being polymorphic, and Lua is a great example that shows having only associative arrays (cf. section 3) is practical. However, we consider the usage of maps and sequences as sufficiently distinct to warrant the distinction between them.

Thus, we propose to include *sequences*, *maps*, and *sets*.

Language Style. Generally, we consider the language style as a given from the host language and thus, it is not part of this discussion. However, aspects such as typing can have a major influence on the design. For dynamic languages, we observed the inclusion of specific collection types for primitive data. Since this increases the number of explicit choices one has to consider, we would argue that it is better to rely on optimizations at the implementation level for use cases where performance is the main driver (cf. section 7).

For typed languages, there seems to be a high degree of exposed complexity for users. For example in Scala, the collection library is designed so that operations on collections produce output collections with the same type as the input collection [24]. This leads to an exposure of highly complex type signatures to the user. Scala choses to mitigate this by including simplified type signatures in its documentation. Thus, we advice collection designers to consider this issue and ensure that tooling and documentation hide such accidental complexity.

Properties. Because of the various properties collections might want to support, it seems best to decide on the most flexible default case and additionally consider mechanisms to opt into or out from certain properties.

A good example is whether to offer fixed or variable sized collections. Variable sized ones seem to be the most flexible solution. However, to facilitate the use case and optimizations for fixed-size collections, it is useful to offer for instance constructors that allow creating a collection with the desired size and default value. Special purpose collections such as bounded queues could then be provided as external libraries.

Similarly, maintaining insertion order for all collections seems to be a choice that guarantees deterministic behavior and thus is often preferable. Other orderings could be offered with operations or iteration constructs. This also means that sorting is arguably something one wants to opt into, for instance by requesting a collection to be sorted or using operations that maintain sorting explicitly.

On the other hand, the choice between having mutable and immutable collections is likely tightly bound to the language style. In the interest of maintaining a minimal set of collections only, deciding on either mutable or immutable seems to be preferable and avoid confusion and duplication. Similarly, we would relegate read-only wrappers or copy-on-write collections to external libraries.

Thread-safety is desirable for languages that support sharing collections between multiple threads. As discussed in section 7, we think this can be provided implicitly without drawbacks. However, to provide atomicity of the right granularity, collections need high-level operations such as `computeIfPresent` or `putIfAbsent`, which typically check some condition on the collection, potentially perform a user-specified operation, and then modify the collection.

To keep the set of collections minimal, parallel execution of bulk operations is best introduced by orthogonal means. This could mean as part of operations for internal traversal or mechanisms for external traversal, possibly using streams.

Whether to support strict or lazy operations seems to be a question of language style and the alternative seems to be a candidate for external libraries.

Weakly referencing collections have many important applications, but remain special purpose, thus, should be part of an external library as well.

Identity-based maps and sets are ideally realized by parameterizing the collection. Ideally, it has defaults appropriate for the language, which can be easily customized.

Operation Design. For a large part, we assume that language style and type dictate a certain operation style. However, operations on collections might be especially easy to access for instance with good code completion. Thus, supporting a wide range of internal iteration operations seems useful and can support complex queries. Concepts such as loops or streams for external iteration can still be beneficial, too. We also argue that these operations are easier to discover than operations hidden in some complex hierarchy of special purpose collection classes. Thus, we consider a small set of collection types with a large number of operations as a choice beneficial for exploratory programming. Furthermore, ensuring a high degree of polymorphism between collections seems especially desirable in the exploratory stage, because it allows switching between collections or generalizing code without accidental complexity and technical issues distracting form the problem to be solved.

Algorithms, Data Structures, and Implementation Choices. Having to pick any specific algorithm or internal representation of a collection in the exploratory programming stage

seems to be solely a distraction. While there can be important performance difference, we would argue that it is better to forgo perfect performance and instead expect the underlying implementation to be *sufficiently smart*. As argued in section 7, it seems feasible and practical to make this tradeoff.

6.2 Discussion

Based on the analysis above, we argue that a collection library designed for exploratory programming should have sequences, maps, and sets as basic abstractions. Relevant properties and modes of operations for collections should be realized by mechanisms that allow for an easy opt in, without requiring additional collection types. In many cases this likely means that the basic collections have a large and versatile set of operations. We assume this can be supported by tooling to provide the desired feedback for instance as part of code completion. However, this also means that many specialized collections should be relegated into external libraries. Depending on the language ecosystem, this hopefully implies only minor inconvenience and is supported by packaging and dependency management systems.

One could now argue that having many different types of operations on the same collection type violates the good design practice of having one thing do one thing alone and do it well. One might even consider such large collection types as an instance of *the blob* anti-pattern. [6]

The strongest argument we see is that code interacting with the collection is not explicit about the desired usage. For instance, a sequence can be easily used as a stack, queue, deque, or list. Other arguments, for instance by Peyton Jones [26] include that such generic types and the lack of specific invariants make it harder to understand programs, harder to prove properties about them, and harder to maintain them. However, in our experience, versatile collections such as JavaScript or Ruby `Arrays` are in most cases used with only a few different operations that clearly indicate a specific "type". For instance, when `push` and `pop` operations are used on a sequence, it is very clear it is meant to be used as a stack. Furthermore, we argue that advanced type inference and run-time feedback can enable development environments to detect such patterns and provide useful feedback to programmers. For languages such as Java, it is idiomatic to use `List` or `Map` instead of the more specific `ArrayList` or `HashMap` to facilitate reuse. This illustrates that in many cases the specific type is less desirable than some might argue.

Furthermore, performance-driven choices are rarely possible in an informed manner during exploratory programming. Having not yet fully understood the problem, one would likely mispredict the distribution of operations at run time. Therefore, specialized classes are unlikely to be beneficial.

Another argument against versatile collections is that their implementation is much more complex. However, the complexity is moved into the runtime, and thus benefits many users. At run time, optimization can become possible without requiring programmer input, as discussed in the following section. For such optimizations, a runtime needs to make however many assumptions, which likely reduces performance predictability. Here we advocate for better tools. Ideally, the environment can utilize the knowledge about run-time optimizations to provide users with information on performance issues, which can be considered by programmers at a point when they start to care about performance.

7 TECHNIQUES FOR EFFICIENT IMPLEMENTATIONS

While we consider performance a secondary concern for exploratory programming, good performance is still essential to make a collection design practical.

With the proposed design, it is likely not possible to achieve 100% of the performance that specialized collections can provide for specific use cases. However, trading some performance for more programmer productivity seems beneficial.

The main technique enabling good performance, even when specialized collection types are not exposed, are to use automatic data structure selection or adaptive data structures. Peyton Jones [26, sec. 2.3] and Chuang and Hwang [9] already contemplated their use. More recently, De Wael et al. [15] experimented with so-called just-in-time data structures, which allows a data structure to specialize based on observed usage patterns. Such techniques are used successfully for collections in dynamic languages. For example, storage strategies [4, 10] ensure that homogeneous collections use efficient in-memory representation avoiding boxing without static types. Statically-typed languages can use mechanisms such as C++ templates to specialize code at compile time.

Xu [30] and Costa and Andrzejak [12] go beyond simply avoiding boxing. They show that collections can be adapted further to take concrete usage in terms of used operations into account. For example, if a `contains()` operation is used frequently on large lists, it can be beneficial to change its implementation to include a hash table to speed up the lookup. For languages that prefer immutable collections, there is similar work for instance by Pape et al. [25], which improves the memory representation of immutable data structures.

As mentioned in section 6.1, thread-safety is ideally provided implicitly without requiring programmer intervention. Furthermore, there should not be any cost associated with such thread-safety if a collection is used only by a single thread. We think this is possible by combining our techniques for thread-safe object representations [14] for dynamic languages with storage strategies.

To gain the last bits of performance, one might want to consider the needs of specialized collections, for instance by ensuring that operations conform to a common interface. In such cases, the instantiation of a generic collection could simply be replaced with a specialized one, possibly from an external library. This might even be facilitated by tooling which determines that a specific sequence allocation results in objects being always used as a queue. At this point, a compiler could select a queue implementation. In gradually-typed languages, this could be further supported by adding types where performance becomes relevant, which may facilitate the selection of a more efficient implementation when only a subset of operations is used.

8 RELATED WORK

While collection design seems to be an integral aspect of language design, we are not aware of any systematic treatment. While there is some work on addressing certain issues [3, 8, 11, 16], there does not seem to be any general discussion of design tradeoffs. However, Odersky and Moors [24] describe their experience redesigning the collection library in Scala. Their focus is mostly on how they achieved an implementation and design that is more principled, structured, and avoids code duplication to avoid bit rot during future maintenance. Also somewhat related, Matthes and Schmidt [21] investigate the question of how to design *bulk types*, i.e., collection libraries, on the intersection to database systems and database programming languages. They discuss the benefits and drawbacks of builtin or library-based designs and favor library-based designs for their extensibility. For languages, we argue for a small set of versatile builtin collections that allows extensions from external libraries.

9 CONCLUSION

This paper informally reviews collection libraries of 13 languages to identify design dimensions and concerns that impact programming experience. Based on the identified dimensions and a brief literature review of collection usage studies, we argue that a small set of collections with versatile operations is beneficial for exploratory programming. Specifically, we propose to only offer three collections, a sequence, a map, and a set. We further argue that these collections can be implemented efficiently with modern run-time techniques.

From our perspective, this design facilitates exploratory programming, because it avoids many typical implementation decisions or relegates them to the runtime. Specifically, it avoids deciding upfront on a specific collection type or data structure. Consequently, adapting the usage of a collection, e.g., from a vector to a stack becomes trivial and does not interrupt the process of understanding the domain problem. Having a large set of operations on a small set of collections also provides better discoverability, because specialized operations are not lost in complex hierarchies of collection types. Thus, the collection library is likely simpler and more convenient to use.

The drawback is that such collections add complexity to language implementations, which must provide complex optimizations to achieve a performance similar to specialized collections. However, many techniques already exist to optimize these collections and increasing language implementation complexity for improved productivity seems to be an acceptable tradeoff.

Future work should chart the design space for collection libraries more thoroughly and completely. It would be useful to develop a common taxonomy for all related concepts because there is rarely any agreement on naming or semantics between different libraries. This is only a first snapshot of a small set of languages and libraries. For example, the ecosystems of Java and C/C++ offer many other collection libraries, with various tradeoffs. Other types of languages, such as array programming, scientific computations, or large-scale parallelism come with their own requirements, which have not been considered.

Further work should also investigate the human component and determine whether our design is indeed beneficial for productivity. Similarly, work on improved collection implementations, a better understanding of performance tradeoffs, as well as support for feedback from development environments is desirable. To provide appropriate feedback, code completion should for instance know how operations relate to each other.

REFERENCES

[1] Miltiadis Allamanis and Charles A. Sutton. 2013. Mining Source Code Repositories at Massive Scale using Language Modeling. In *10th Working Conference on Mining Software Repositories (MSR'13)*. IEEE, 207–216. https://doi.org/10.1109/MSR.2013.6624029

[2] Alexandre Bergel, Alejandro Infante, Sergio Maass, and Juan Pablo Sandoval Alcocer. 2018. Reducing Resource Consumption of Expandable Collections: The Pharo Case. *Science of Computer Programming* (2018). https://doi.org/10.1016/j.scico.2017.12.009

[3] Andrew P. Black, Nathanael Schärli, and Stéphane Ducasse. 2003. Applying Traits to the Smalltalk Collection Classes. In *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programing, Systems, Languages, and Applications (OOPSLA'03)*. ACM, 47–64. https://doi.org/10.1145/949305.949311

[4] Carl Friedrich Bolz, Lukas Diekmann, and Laurence Tratt. 2013. Storage Strategies for Collections in Dynamically Typed Languages. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA'13)*. ACM, 167–182. https://doi.org/10.1145/2509136.2509531

[5] Gilad Bracha, Peter von der Ahé, Vassili Bykov, Yaron Kashai, William Maddox, and Eliot Miranda. 2010. Modules as Objects in Newspeak. In *ECOOP 2010 – Object-Oriented Programming*. Lecture Notes in Computer Science, Vol. 6183. Springer, 405–428. https://doi.org/10.1007/978-3-642-14107-2_20

[6] William J. Brown, Raphael C. Malveau, Hays W. McCormick, and Thomas J. Mowbray. 1998. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis.* John Wiley and Sons.

[7] Sebastian Burckhardt, Manuel Fahndrich, Peli de Halleux, Sean McDirmid, Michal Moskal, Nikolai Tillmann, and Jun Kato. 2013. It's Alive! Continuous Feedback in UI Programming. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'13)*. ACM, 95–104. https://doi.org/10.1145/2491956.2462170

[8] Manuel M. T. Chakravarty, Gabriele Keller, Simon L. Peyton Jones, and Simon Marlow. 2005. Associated Types with Class. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'05)*. ACM, 1–13. https://doi.org/10.1145/1040305.1040306

[9] Tyng-Ruey Chuang and Wen L. Hwang. 1996. A Probabilistic Approach to the Problem of Automatic Selection of Data Representations. In *Proceedings of the First ACM SIGPLAN International Conference on Functional Programming (ICFP'96)*. ACM, 190–200. https://doi.org/10.1145/232627.232648

[10] Daniel Clifford, Hannes Payer, Michael Stanton, and Ben L. Titzer. 2015. Memento Mori: Dynamic Allocation-site-based Optimizations. In *Proceedings of the 2015 International Symposium on Memory Management (ISMM'15)*. ACM, 105–117. https://doi.org/10.1145/2754169.2754181

[11] William R. Cook. 1992. Interfaces and Specifications for the Smalltalk-80 Collection Classes. In *Conference Proceedings on Object-oriented Programming Systems, Languages, and Applications (OOPSLA'92)*. ACM, 1–15. https://doi.org/10.1145/141936.141938

[12] Diego Costa and Artur Andrzejak. 2018. CollectionSwitch: A Framework for Efficient and Dynamic Collection Selection. In *Proceedings of 2018 IEEE/ACM International Symposium on Code Generation and Optimization (CGO'18)*. 16–26. https://doi.org/10.1145/3168825

[13] 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 (ICPE'17)*. ACM, 389–400. https://doi.org/10.1145/3030207.3030221

[14] Benoit Daloze, Stefan Marr, Daniele Bonetta, and Hanspeter Mössenböck. 2016. Efficient and Thread-Safe Objects for Dynamically-Typed Languages. In *Proceedings of the 2016 ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA'16)*. ACM, 642–659. https://doi.org/10.1145/2983990.2984001

[15] Mattias De Wael, Stefan Marr, Joeri De Koster, Jennifer B. Sartor, and Wolfgang De Meuter. 2015. Just-in-Time Data Structures. In *Proceedings of the 2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (Onward!'15)*. ACM, 61–75. https://doi.org/10.1145/2814228.2814231

[16] Ronald Garcia, Jaakko Järvi, Andrew Lumsdaine, Jeremy Siek, and Jeremiah Willcock. 2007. An extended comparative study of language support for generic programming. 17, 2 (2007), 145–205. https://doi.org/10.1017/S0956796806006198

[17] Adele Goldberg and David Robson. 1983. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[18] Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes. 2005. The Implementation of Lua 5.0. *Journal of Universal Computer Science* 11, 7 (2005), 1159–1176. https://doi.org/10.3217/jucs-011-07-1159

[19] ISO/IEC. 2017. *Programming Languages — C++.* Draft International Standard N4660. https://web.archive.org/web/20170325025026/http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/n4660.pdf

[20] Stefan Marr and Hanspeter Mössenböck. 2015. Optimizing Communicating Event-Loop Languages with Truffle. (26 October 2015).

[21] Florian Matthes and Joachim W. Schmidt. 2000. *Bulk Types: Built-in or Add-On?* Springer, Berlin, Heidelberg, 257–261. https://doi.org/10.1007/978-3-642-59623-$0_1$6

[22] Sean McDirmid. 2007. Living It Up with a Live Programming Language. In *Proceedings of the 22Nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications (OOPSLA'07)*. ACM, 623–638. https://doi.org/10.1145/1297027.1297073

[23] Erik Meijer. 2011. The World According to LINQ. *Commun. ACM* 54, 10 (Oct. 2011), 45–51. https://doi.org/10.1145/2001269.2001285

[24] Martin Odersky and Adriaan Moors. 2009. Fighting bit Rot with Types (Experience Report: Scala Collections). In *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (Leibniz International Proceedings in Informatics (LIPIcs))*, Ravi Kannan and K. Narayan Kumar (Eds.), Vol. 4. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 427–451. https://doi.org/10.4230/LIPIcs.FSTTCS.2009.2338

[25] Tobias Pape, Carl Friedrich Bolz, and Robert Hirschfeld. 2017. Adaptive Just-in-time Value Class Optimization for Lowering Memory Consumption and Improving Execution Time Performance. *Science of Computer Programming* 140 (2017), 17–29. https://doi.org/10.1016/j.scico.2016.08.003

[26] Simon L. Peyton Jones. 1996. Bulk types with class. In *Proceedings of the Second Haskell Workshop.* https://www.microsoft.com/en-us/research/publication/bulk-types-with-class/

[27] Patrick Rein, Stefan Lehmann, Toni Mattis, and Robert Hirschfeld. 2016. How Live Are Live Programming Systems?: Benchmarking the Response Times of Live Programming Environments. In *Proceedings of the Programming Experience 2016 (PX/16) Workshop (PX/16)*. ACM, 1–8. https://doi.org/10.1145/2984380.2984381

[28] Steven L. Tanimoto. 1990. VIVA: A Visual Language for Image Processing. *Journal of Visual Languages & Computing* 1, 2 (1990), 127 – 139. https://doi.org/10.1016/S1045-926X(05)80012-6

[29] Steven L. Tanimoto. 2013. A Perspective on the Evolution of Live Programming. In *Proceedings of the 1st International Workshop on Live Programming (LIVE'13)*. IEEE, Piscataway, NJ, USA, 31–34. https://doi.org/10.1109/LIVE.2013.6617346

[30] Guoqing Xu. 2013. CoCo: Sound and Adaptive Replacement of Java Collections. In *ECOOP 2013 – Object-Oriented Programming: 27th European Conference, Montpellier, France, July 1-5, 2013. Proceedings*, Giuseppe Castagna (Ed.). Springer, 1–26. https://doi.org/10.1007/978-3-642-39038-8_1