THE UNIVERSITY of EDINBURGH
informatics

Collection Skeletons: Declarative Abstractions for Data Collections[a][b]

7th UK System Research Challenges Workshop

Björn Franke[1], **Zhibo Li**[1], Magnus Morton[2], Michel Steuwer[1]

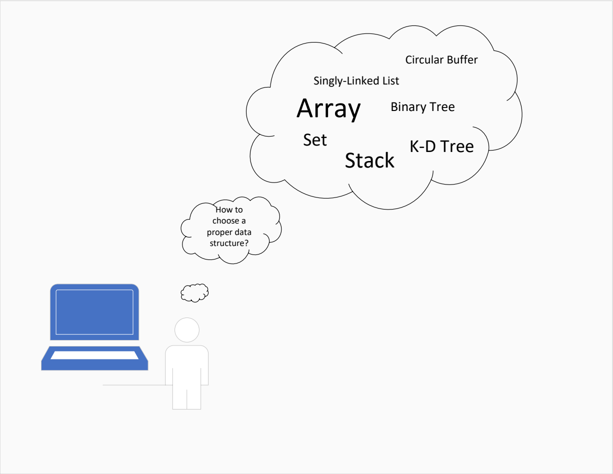
27/04/2023

[1]University of Edinburgh
[2]Huawei Central Software Institute

[a]Funded by the Huawei-Edinburgh Joint Lab
[b]Best Paper Award, SLE 2022

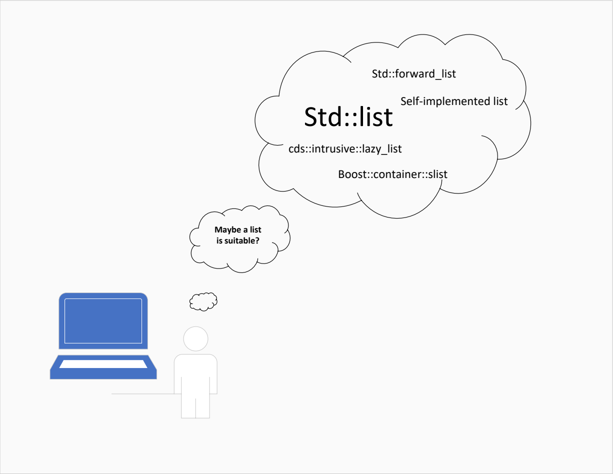
Collections in Programming

Collections are of vital importance in programming. In practice, to choose the right collection and implementation given a problem domain and target platform is not easy.



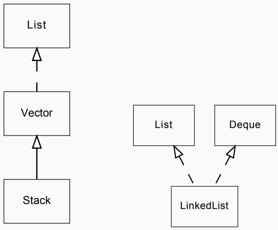
Collections in Programming

Let us stick with a *list*, but there are many implementation choices.



Collections in Programming

It can be equally hard for a programmer to even select the most appropriate collection from an existing collection hierarchy[1].

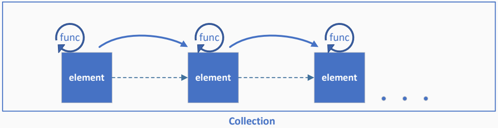


Why would a stack *be* a vector or a list?

[1] Chengpeng Wang, Peisen Yao, Wensheng Tang, Qingkai Shi, and Charles Zhang. Complexity-guided container replacement synthesis. OOPSLA 2022.

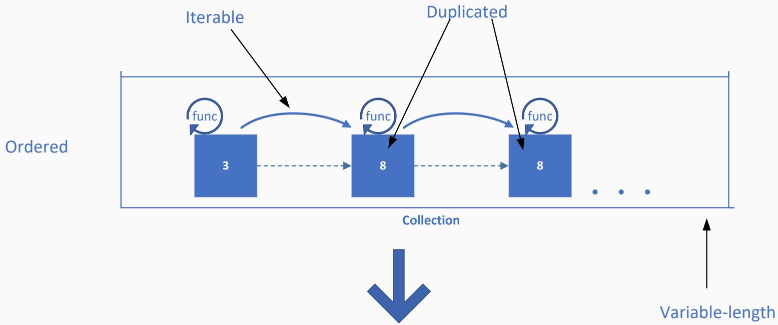
How to Choose A *Proper* Data Collection

Let us consider a simple example. We would like to write a piece of code consisting of a **loop traversing** over a data collection of integers one by one. For each element we would like to increase the element value by 1.



What data collection shall we use?

Let us go over the problem domain -



Collection<Integers,Iterable,Ordered,Duplicated,Variable-length>

Collection Skeletons

Collection Skeletons

We develop Collection Skeletons which provide a novel, **declarative** approach to data collections.

- Exposing individual properties to be specified (rather than implicit properties like in ADTs)
- Identify a set of useful semantic and interface properties, which capture the key aspects of data collections programmers care about
- Evaluate a prototype C++ library implementation of our Collection Skeletons framework and demonstrate negligible performance impact across three different hardware platforms

Properties

We propose **eight** groups of properties to model our Collection Skeletons. We distinguish between:

- **Semantic properties**
- **Interface properties**
- Future Work: **Non-Functional properties:** runtime, space, ...

Semantic Properties

Semantic properties specify the **behaviour** of the collections and methods with which collections are accessed or modified.

- **Uniqueness**, e.g. a *set* – behaviour of *insert* function
- **Circularity**, e.g. a *circular buffer* – behaviour of *next* function

Interface properties

Interface properties specify certain **functionality**, usually in form of access methods to be provided by the collection.

- Variability - *append()*
- Iterable - *iter()*
- Accessibility - *[]* operator
- Splitability - *splitAt()*
- UnionFind - *union()* and *find()*

Hybrid Properties

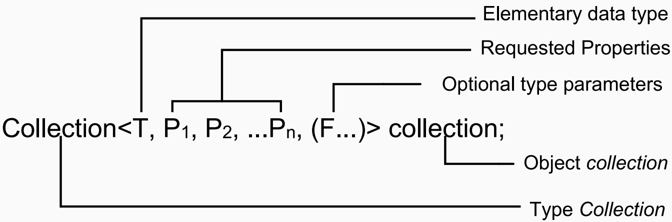
Some properties are of **hybrid** nature, i.e. they both specify **access methods** and also change the way operations behave **semantically**. An example of such a hybrid property is order

- **Order** - e.g. *FIFO*, *Ordered*
  Provides `iter()` and changes the behaviour of the iterator

C++ Library & API

We have implemented a prototype library with C++ template meta programming for our Collection Skeletons.

Declaring a collection is done like this:

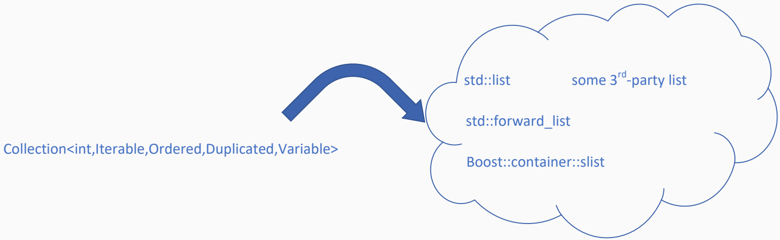


Implementation of the Prototype Library

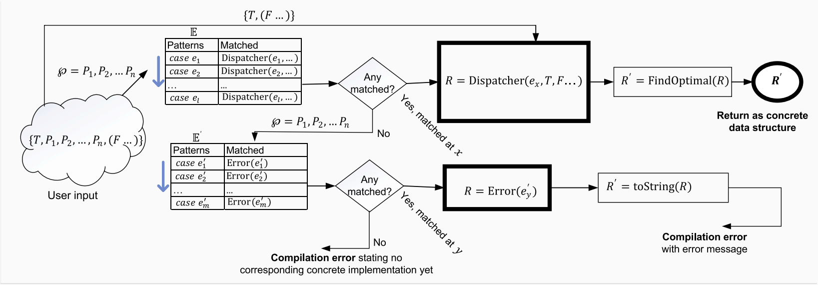
Based on the programming interface, we develop

- *member functions* for corresponding *interface properties*
- *default methods* for all collections

How to map the property-based declaration to the concrete data structure implementation?



Multi-Staged Pattern Matching

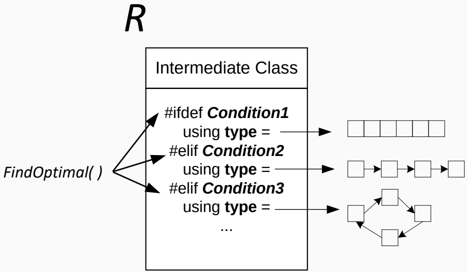


Pattern matching - possible outcomes:

1. Properties form an *eligible declaration*
   1.1 Concrete data structure found in library
   1.2 Failure to find concrete data structure - library incomplete, error
2. Properties form an *erroneous declaration* - compilation error

Selecting A Concrete Data Structure from A Candidate List

Candidate List - *R* is an *intermediate class* that enables implementation flexibility and the selection of an optimal concrete data structure.



Can be automated, e.g. CollectionSwitch, CGO'18.

API Design and Rules

Simple and user-friendly API, but rules are required to prevent nondeterministic behaviours.

- The property list is order-free
  Collection<A,B> = Collection<B,A>
- Mutually exclusive properties cannot co-exist in a property list
  Collection<A,¬A>
- *No guarantee on algebraic operations for properties*
  Collection<?> = merge(Collection<A>,Collection<B>)

Collections & Parallelism

Types of Parallelism

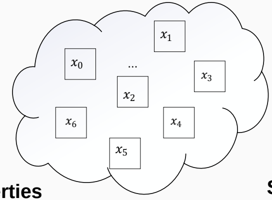
- *Implicit* Parallelism
  Transparent to the applications programmer
  Hidden in e.g. collection access functions
- *Explicit* Parallelism
  Exposed to the applications programmer
  Parallel algorithmic skeletons

Implicit Parallelism

- Hiding parallel implementations “behind the scenes”, e.g. parallel STL
- Parallelism is encapsulated within e.g. *find* operation
- Just another collection implementation in our scheme
- No difference in application code

Collections & Algorithmic Skeletons

$map(\text{collection}, f)$

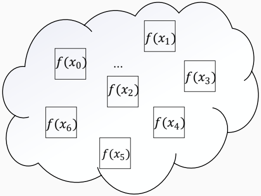


Necessary properties
for a map

iterable – $iter(), next()$

Sufficient properties
for a **parallel** map

random accessible – [ ]



Collections & Algorithmic Skeletons

- Data-Parallel Skeletons – *iterable*, *random access*
- Divide & Conquer – *splitable*, by value or position; *mergeable*
- Stencil – *neighbourhood* property, provides neighbourhood collection for each element; *Rectangular/Square*
- Wavefront – *frontier*, based on data dependence properties of *f*!

Collections & Algorithmic Skeletons

Interesting questions:

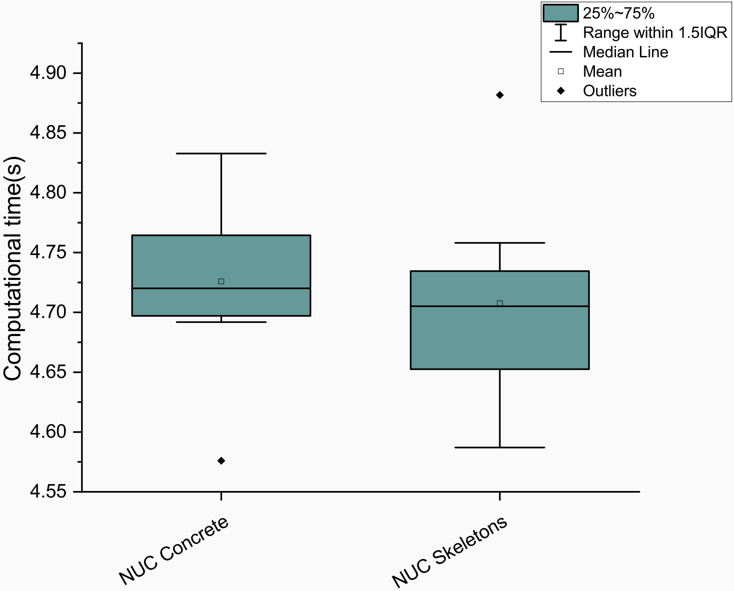
- What properties can be automatically inferred from source code? e.g. interface properties
- What properties can be inferred for the resulting collection?
  - if properties of the source collection are known
  - if properties of the function are known, e.g. injective
- What are the rules of collections?
  e.g. merging two collections w.r.t. resulting properties
- Can we check properties of C & f at compile-time/run-time?
- Is our set of properties complete?

Evaluation

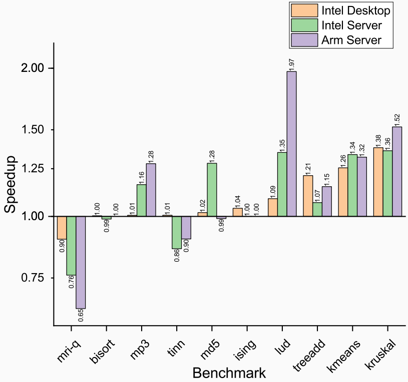
Benchmarks and Benchmarking Methodology

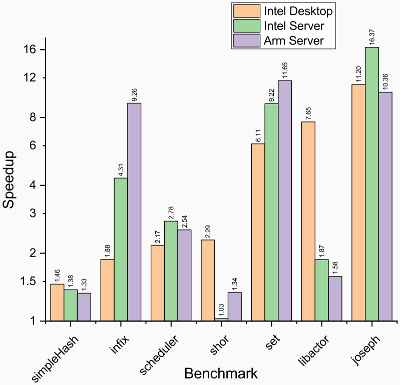
- Prototype library evaluated against standard benchmarks (Olden, Rodinia, Parboil), open-source applications, and micro-benchmarks.
- Manual rewriting of legacy C benchmarks
    - replaced existing low-level data structures and their access functions,
    - no other code rewriting/optimisation,
    - same input data for all versions for performance comparison.
- Three target platforms (Intel Desktop, Intel Server, Arm Server)

Experimental Results - Abstraction Overhead



Experimental Results - Speedup





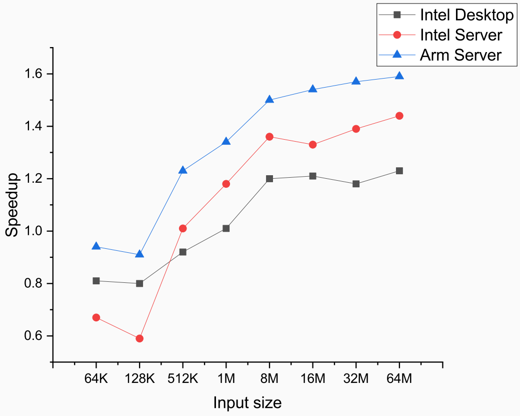
Experimental Results - Flexibility

Optimal concrete data structure for each benchmark and target platform

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

Experimental Results - Performance Influencing Factors

Some factors, such as the size of the data collection, can have an impact on its performance and different architectures may offer different performance trade-offs.



Speedup of array-based binary tree for bisort over a range of dataset sizes

Experimental Results - Implicit Parallelism

Benchmark	Details	Speedup Nuc	Speedup Arm	Speedup Server
ising	concurrent vector	5.04	4.53	32.22
set	concurrent vector	1.62	1.52	1.02
tinn	concurrent vector	NP	1.01	NP
mp3	concurrent vector	NP	NP	NP
scheduler	concurrent_priority_queue	NP	NP	NP
md5	concurrent_unordered_map	NP	NP	NP
infix	lock free stack lock free queue	NP	NP	NP
kruskal	concurrent_unordered_map	NP	NP	NP

Experimental Results - Parallel Algorithmic Skeletons

Benchmark	Details	Speedup Nuc	Speedup Arm	Speedup Server
ising	for_each	5.04	4.53	32.22
libactor	for_each	1.03	NP	1.13
tinn	map, reduce, zip	NP	1.01	NP
simpleHash	for_each	NP	1.78	NP
lud	map reduce	NP	NP	NP
kmeans	map reduce	NP	NP	NP
mri-q	map reduce zip	1.12	1.09	1.17

Summary, Conclusions & Future Work

Summary & Conclusions

- *Declarative data collections* exposing *fundamental collection properties* to the programmer - simplification for the user
- No/minimal performance overhead
- Opportunity for *performance improvements* through greater implementation flexibility
- *Different optimal* concrete data structures depend on application context and target platform
- Scope for implicit and explicit parallelism – compatibility with parallel algorithmic skeletons
- *Speedups* across a range of benchmarks for different target platforms - average speedup 2.57-2.93.

Future Work

- Wider range of supported platforms, e.g. GPUs and accelerators
- More on Collection Skeletons and Parallel Algorithmic Skeletons
- Dynamic adaptation at runtime
- Other problem domains: matrices, graphs,… with *dynamic/data-dependent* properties

Thanks for listening! Questions?