Auto-Collections

Allan Renucci
allan.renucci@epfl.ch

School of Computer and Communication Sciences

Optional Semester Project Report

Supervisor
Dmytro Petrashko
EPFL / LAMP
dmitry.petrashko@epfl.ch

Supervisor
Prof. Martin Odersky
EPFL / LAMP
martin.odersky@epfl.ch

June 2016

Auto-Collections

Allan Renucci
EPFL, Switzerland
allan.renucci@epfl.ch

Abstract

Modern programming languages such as Scala, Java and C# make extensive use of collections.

A collection implementation represents a fixed choice in the dimensions of operation time and space utilization. Using the collection in a manner not consistent with this fixed choice can cause significant performance degradation.

Often programmers have to choose between under-performing generic collection and specific data-structures at the cost of modularity and extensibility.

In this paper, we present and evaluate a generic collection library that automatically chooses the appropriate collection implementations for an application.

We implemented our solution in the Dotty compiler for Scala and showed that our implementation can choose optimal collections for common basic algorithms.

1 Introduction

Programs are a combination of algorithms and data-structures [7]. Performance of program and performance of data-structures are inherently related. For example, if one manipulates data in a last-in-first-out order, one will use a data-structure optimized for this use-case such as a Stack. Choosing non-optimal data-structures may incur significant performance costs. Software algorithms require input data and produce output results. As a rule, these input/output data representations are closely tied to a given algorithmic implementation and hence impose limitations on modularity and extensibility. Often programmers have to choose between under-performing generic data-structures and specific data-structures at the cost of modularity and extensibility.

In this report, we propose and evaluate an alternative collections library designed for good performance without sacrificing modularity and extensibility. The user can choose between three abstract data types (i.e. `Seq`, `Map` and `Set`) and the compiler will automatically select an optimal data structure implementation from a given use case. The compiler selects the appropriate collection

implementation based on instance profiles (e.g. elements' type, set of methods called on one instance) collected during call-graph construction.

Modern programming languages make extensive use of collection classes. Although, we will present an implementation for Scala inside the Dotty Compiler, it is conceivable to apply the proposed techniques to other languages that have abstraction features such as inheritance and generics. The report presents experimental results showing that our implementation can choose optimal collections for common basic algorithms.

The rest of the report is organised as follow. In Section 2, we motivate the use of automatic collections. In Section 3, we explain the basic algorithm behind auto-collections. Section 4 describes the implementation inside the Dotty compiler. Section 5 presents and discusses our experimental results. We discuss related work in Section 6 and conclude in Section 7.

2 Motivation

Correct choice of data-structure is hard A programmer's choice of data structure ultimately determines which operations on a data-set will be efficient, meaning the operation can complete within the time and storage constraints imposed. Every data structure represents a particular trade-off between time and storage space, making some operations faster (or less space-consumptive) and some operations slower. An ordered list makes finding the smallest element fast, but inserting new elements slow. A hash set allows for quick insertions and retrievals of specific items, but finding the smallest element is slow. Understanding these trade-offs and selecting a data structure appropriate for the application at hand is hard. Thus, freeing the user from managing and choosing the right data structure for their application is particularly relevant.

Correct choice of data-structure breaks modularity All algorithms require input data and the results of algorithms are also stored in data structures. Modularity and extensibility demands that algorithmic implementations be independent from these input and data-structures. Indeed, if one specifies the most specific type of collection that works best in his tests / his intention, he is going to hit users if his assumptions are proved wrong. What library authors do instead now is using collections that decently support all operations, which in turn means that it is under-performing in every particular use cases. With automatic collections chosen based on the use cases, one would not need to trade modularity for performance.

3 Algorithm

Now that we justified the need for automatic collections, our goal is to find an assignment of collection implementations that is optimal for a given program. An optimal choice of collection implementations tries to balance two dimensions:

minimizing the time required to perform operations while also minimizing the space required to represent application data. Our approach is to select collection implementations based on collection usage statistics extracted from the client program. Here is a non-exhaustive list of statistics that can be used during the decision process:

- The set of methods called on a collection instance. We want to find the collection that minimizes the cost of each operations.
- The type of underlying elements. There exists specific implementation that are optimised for a particular type of element (e.g. LongMap).
- The presence of pattern in the underlying elements. For example, if all element are numbers with a constant delta value, we can use a Range.
- The operation counts. We want to emphasize on operations that are performed multiple time, inside loops or inside recursive functions.

Although, this information can be obtained either statically or dynamically, we choose to do it statically at compile time which has the advantage of not inducing an overhead at runtime.

4 Implementation

We implemented our solution inside the Dotty Linker[1], a modified version of the Dotty Compiler[2] supporting language specific and library-specific optimizations. Dotty is a new compiler under active development for the future evolution of the Scala language.

Our implementation relies on call graph construction [2]. A call graph is a directed graph that represents calling relationships between functions in a computer program. We use the call graph to track the method called on a specific collection instance. We proceed in two steps:

1. In a first phase preceding call graph construction, we substitute each auto-collection creation site by the creation of a new anonymous class which defines a unique type and uniquely identifies a collection instance.
2. In a second phase after call graph construction, we traverse the call graph and collect all the methods called on each auto-collection instances now identified by a unique type.

   We can then pick an optimal collection implementation based on the set of methods called on this instance and the type of the underlying elements.

Consider the example in Figure 1 describing the successive transformations performed by the compiler. The code written by the user on Line 2 creates

[1] `https://github.com/dotty-linker/dotty`
[2] `https://github.com/lampepfl/dotty`

```
1  // User code
2  val seq = AutoSeq[Int](1, 2, 3)
3  val h   = seq.head
4
5  // After first transformation
6  val seq: Seq[Int] = {
7      class AnonSeq() extends Seq[Int] { ... }
8      new AnonSeq()
9  }
10
11 // After second transformation
12 val seq: Seq[Int] = Queue[Int](1, 2, 3)
```

Figure 1: Compiler Transformations

a new automatic sequence of integers. During the first transformation, the compiler substitute the code on Line 2 by the code on line 6, 7, 8 and 9: a unique anonymous class which extends the `Seq` abstract type is created as well as an instance of this class. At this time, the compiler builds a call graph for the program and makes it available to later phases. On line 12, the compiler finally substitutes the previously generated code by an optimal implementation of sequence for this program.

Table 4 lists the supported collection implementations as well as the conditions under which one implementation is chosen over the others. Conditions are listed by precedence order. If multiple conditions match then the first one will be chosen.

At the moment, our solution is limited by the amount of information available during the decision process. Indeed, we would benefit from information such as the number of times a function is called, if a function is called inside a loop or a recursive function. Then, one could give more importance to one function other the others and the decision process would be improved. Such information can be extracted via data-flow analysis [1] or pointer analysis [3].

5 Evaluation

We evaluated our solution on common basic algorithms and made sure that it selected optimal data-structures.

QuickSort QuickSort sorts a sequence in-place and does not use additional data-structures but the sequence to be sorted. However QuickSort performs several random accesses on the sequence and the performance of the algorithm is inherently related to the cost of these accesses.

In our experiment, our solution picked an array for the collection to be sorted as it offers the best performance for random accesses.

Type	Semantic	Condition	Implementation
Seq	Immutable	head and tail are the only operations	List
		elements are appended or prepended	immutable.Queue
		operations are by indexes and elements' type is Char	WrappedString
		operations are by indexes	WrappedArray
		elements are integers with a constant delta	Range
		default immutable Seq	Vector
	Mutable	operations are by indexes	WrappedArray
		elements' type is known	UnrolledBuffer
		default mutable sequence	ListBuffer
Map	Immutable	elements are added	immutable.HashMap
		keys' type is subtype of AnyRef	mutable.AnyRefMap
		keys' type is Long	LongMap
		default immutable Map	mutable.HashMap
	Mutable	default mutable Map	mutable.HashMap
Set	Immutable	default immutable Set	immutable.HashSet
	Mutable	default mutable Set	mutable.HashSet

Table 1: Supported Collection Implementations

MergeSort We implemented the top-down MergeSort algorithm which recursively divides the input sequence into smaller sub-sequences until the sub-sequences are trivially sorted, and then merges the sub-sequences while returning up the call chain. The algorithm needs a sequence implementation which provides good performances for split, append, head removal and size tests.

In our experiment, our solution picked a queue. It provides constant time append and head removal. Size tests and split take linear time. A queue is a correct choice of data-structure as long as the Scala Standard Library does not provide an optimal collection for this particular use case.

BFS Breadth-first search on a tree maintains a sequence of nodes to visit next. BFS appends nodes to the sequence and removes nodes from the head of the sequence.

In our experiment, our solution picked a queue where both append and head removal run in constant time.

DFS Iterative Depth-first search on a tree also maintains a sequence of nodes to visit next. DFS removes nodes from the head of the sequence but prepends nodes to the sequence as opposed to BFS.

In our experiment, our solution also picked a queue as the queue implementation from the Scala Standard Library provides a constant time preprend operation.

PageRank In our implementation of PageRank, we initially start with a map from page to their score and iteratively improve the score until convergence of the algorithm. The page need to be uniquely identified but its type is arbitrary.

In our experiment, we computed PageRank using Long as unique identifiers for pages. Our solution chose a LongMap which is a specialised implementation of Map for Long keys. It provides significantly faster operation than the default map implementation (i.e. HashMap).

6 Related Work

There exists several tools that assist the programmer in choosing the appropriate collection implementations for his application.

One of them is called CHAMELEON [5]. CHAMELEON computes elaborate trace and heap-based metrics on collection behavior. These metrics are consumed on-the-fly by a rules engine which outputs a list of suggested collection adaptation strategies. The tool can apply these corrective strategies automatically or present them to the programmer. As opposed to our solution which collects information on collection at compile time, CHAMELEON does it during program execution. CHAMELEON is implemented on top of the JVM and induces an overhead during program execution. Our solution is implemented on top of the Dotty Compiler which induces an overhead at compile time but none at runtime.

Bjorn De Sutter [6] presents another approach for rewriting applications to use customized versions of library classes that are generated using a combination of static analysis and profile information. Type constraints are used to determine where customized classes may be used, and profile information is used to determine where customization is likely to be profitable. This approaches requires static analysis unlike our work where programs are correct by construction. In this setting, static analysis is needed to guarantee type correctness in cases where objects are exchanged with the standard libraries (or other components). This greatly limits the number of possible optimisations and the programmers has to choose collections with this constraints in mind. In our setting, the programmer has access to a generic collection API which exempt him from choosing specific collection implementations. The compiler will take care of choosing an optimal collection for a given use case.

The challenge of freeing the user from managing and choosing the right data structure for their application is not a new one. Schonberg [4] discusses techniques for automatically selecting concrete data structures to implement the abstract data types set and map in SETL programs. Depending on whether or not iterators and set-theoretic operations such as union and intersection are applied to abstract data types, their optimizing compiler selects an implementation from a predetermined collection of implementations in which sets are represented as linked lists, hash-tables, or bit-vectors. Our solution extends this idea by using additional information on the underlying elements as well as the methods applied to a collection.

7 Conclusion & Future Work

We presented our solution for automatic collections that frees the user from managing and choosing the right data structure for their application. It has the benefit of improving program efficiency with no negative impact on modularity and extensibility.

We implemented our solution in the context of the Dotty compiler and showed that our implementation can choose optimal collections for common basic algorithms.

While our work was primarily focused on Scala, the ideas of our work are applicable to other statically typed languages that have abstraction features such as inheritance and generics such as Java, C#, C++, Haskell, Swift, and D.

In the future, we plan to support more collections from the Scala Standard Library as well as customized version of library classes. We also plan to improve our decision process with more collection usage statistics extracted via data-flow analysis.

References

[1] Gary A. Kildall. A unified approach to global program optimization. *Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages - POPL '73*, 1973.

[2] Dmytro Petrshko, Vlad Ureche, Ondrej Lhotak, and Martin Odersky. Call graphs for languages with parametric polymorphism. In *Proceedings of the 2016 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2016, part of SPLASH 2016, Amsterdam, Netherlands, October 30 - November 4, 2016*. ACM, 2016.

[3] Barbara G. Ryder. Dimensions of precision in reference analysis of object-oriented programming languages. *Lecture Notes in Computer Science Compiler Construction*, page 126–137, 2003.

[4] Edmond Schonberg, Jacob T. Schwartz, and Micha Sharir. An automatic technique for selection of data representations in setl programs. *ACM Transactions on Programming Languages and Systems ACM Trans. Program. Lang. Syst. TOPLAS*, 3(2):126–143, Jan 1981.

[5] Ohad Shacham, Martin Vechev, and Eran Yahav. Chameleon. *ACM SIGPLAN Notices SIGPLAN Not.*, 44(6):408, 2009.

[6] Bjorn De Sutter, Frank Tip, and Julian Dolby. Customization of java library classes using type constraints and profile information. *ECOOP 2004 – Object-Oriented Programming Lecture Notes in Computer Science*, page 584–608, 2004.

[7] Niklaus Wirth. *Algorithms + data structures = programs.* Prentice-Hall, 1976.