Towards a Software Product Line of Trie-Based Collections

Michael J. Steindorfer
Centrum Wiskunde & Informatica, The Netherlands
Michael.Steindorfer@cwi.nl

Jurgen J. Vinju
Centrum Wiskunde & Informatica, The Netherlands
Jurgen.Vinju@cwi.nl

Abstract

Collection data structures in standard libraries of programming languages are designed to excel for the average case by carefully balancing memory footprint and runtime performance. These implicit design decisions and hard-coded trade-offs do constrain users from using an optimal variant for a given problem. Although a wide range of specialized collections is available for the Java Virtual Machine (JVM), they introduce yet another dependency and complicate user adoption by requiring specific Application Program Interfaces (APIs) incompatible with the standard library.

A product line for collection data structures would relieve library designers from optimizing for the general case. Furthermore, a product line allows evolving the potentially large code base of a collection family efficiently. The challenge is to find a small core framework for collection data structures which covers all variations without exhaustively listing them, while supporting good performance at the same time.

We claim that the concept of Array Mapped Tries (AMTs) embodies a high degree of commonality in the sub-domain of immutable collection data structures. AMTs are flexible enough to cover most of the variability, while minimizing code bloat in the generator and the generated code. We implemented a Data Structure Code Generator (DSCG) that emits immutable collections based on an AMT skeleton foundation. The generated data structures outperform competitive hand-optimized implementations, and the generator still allows for customization towards specific workloads.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors – Code generation, Optimization; E.1 [*Data Structures*]: Trees

Keywords Code generation, Persistent data structure, Hash trie, Performance, Immutability, Software product line

GPCE'16, October 31 – November 1, 2016, Amsterdam, Netherlands
© 2016 ACM. 978-1-4503-4446-3/16/10...$15.00
http://dx.doi.org/10.1145/2993236.2993251

1. Introduction

Collection data structures that are contained in standard libraries of programming languages are popular amongst programmers. Almost all programs make use of collections. Therefore optimizing collections implies automatically increasing the performance of many programs. Optimizations within collection libraries are orthogonal to compiler and runtime improvements, because they usually focus on improving data structure encodings and algorithms.

Immutable collections represent key data structures in hybrid functional and object-oriented programming languages, such as Scala[1] and Clojure[2]. Immutability allows optimizations that exploit the fact that data does not change [16, 29], allows safe sharing of data in concurrent environments, and makes equational reasoning possible in object-oriented programming environments.

Collection data structures that are contained in standard libraries are mostly one-off solutions, aiming for reasonable performance for the general use case. Design decisions and trade-offs are preselected by the library engineer and turn collection data structures into hard-coded assets. This is problematic, since statically encoding data structure design decisions and trade-offs brings disadvantages for the library users and the library engineers. While the former do not have easy access to optimized problem-specific data structures, the latter cannot extend and evolve potentially large code bases of collection libraries efficiently.

A Large Domain with Variability. The various dimensions of collection libraries in languages such as Java or Scala become apparent when looking at their package structures. They provide many data structure variations that duplicate code and are split by several of the following dimensions:

Split by data type semantics: Interfaces and implementations for lists, sets, bags, maps, multi-maps, etcetera.

Split by ordering: Data structures can be ordered by data type semantics or temporal properties such as insertion order. Otherwise, data structures can be unordered by nature (e.g., sets) or unordered due to hashing of the keys.

[1] `https://scala-lang.org`

[2] `https://clojure.org`

Split by update semantics: Data structures can allow mutation of their content over time, or remain immutable after initialization. Transient data structures represent the middle ground by allowing efficient initialization and batch updates on otherwise immutable data structures.

Split by processing semantics: Data structures are often divided into categories by their supported processing semantics. They can either support basic sequential processing, parallel processing (e.g., by splitting and merging data), or concurrent processing.

Split by encoding: Different encodings yield different performance characteristics. For example, a list data type allows implementations as an array, or as entries that are linked through references.

Split by content: Most collection data structures are designed to be type-safe by restricting elements to a single homogeneous generic type. Storing mixed content of various types is often only possible untyped.

Given the above (incomplete) indication of variability, collection libraries seem like an ideal case for generative programming in the traditional sense [19, 5, 9]. We expect to factor out commonalities for ease-of-maintenance, improve efficiency, and make variants available as context-specific solutions. Because of the large amount of variability, the challenge is to find a minimal core that is expressive enough to cover the domain while at the same time offer good performance. We claim that by fixing the dimension of update semantics to immutable (and transient), we can provide a minimal core, on basis of an Array Mapped Trie (AMT) skeleton, which is able to satisfy our performance requirements.

Without loss of generality, AMTs do allow the generation of mutable collections. However, early experiments showed that these generally exhibit weaker performance characteristics than competing array-based data structures. We limit our motivation and claims in this paper to immutable data.

Contributions. We contribute a domain analysis that covers variability in collection data structures, and the application of AMT skeletons in our domain specific code generator, factoring out commonalities while enabling performance.

2. Related Work

Software Product Lines and Dynamic Adaptation. We take a static Software Product Line (SPL) [8] perspective on collections to enable software reuse. Features of collections and variability are typically known at design time. Dynamic Software Product Lines [13] and Run-Time Adaptation [1] cover variability at program runtime. AMTs are amenable to run-time variability as well; which we consider future work.

Data Structure Selection at Run-Time. SETL pioneered automatic data structure selection [22]. On the Java Virtual Machine (JVM), Shacham et al. [23] introduced Chameleon, a dynamic analysis tool that lets programmers choose the most efficient implementation for a given collection Application Program Interface (API). Regardless of data structure selection, neither SETL nor Chameleon is concerned with our goal of encoding commonalities of a product family of data types.

Generating Complex Collection Data Structures. Declaratively synthesizing complex collection data structures by component composition goes back to DiSTiL [24].

Hawkins et al. worked on declarative and provable specifications and synthesis of data structures with complex sharing, both for the sequential [15] and concurrent [14] case.

Loncaric et al. [18] extend the work of Hawkins et al. by adding support for order among elements and complex retrieval operations. They generate *intrusive* data structures that avoid a layer of indirection by storing auxiliary pointers in domain elements directly, trading flexibility of generic collections for a potential increase in performance. In contrast, our approach natively supports sharing of sub-structures and focuses on non-intrusive collections, however we do not integrate formal methods for making correctness claims.

All previously discussed papers have one approach in common: they synthesize complex data structures by composing basic collection data structures (e.g., array-list, linked-list, hash-map, etcetera). None of these results tackle the generation of basic collection APIs like the current paper does.

Specializing for Primitive Data Types. Ureche et al. [31] added automatic specializations for primitive JVM data types to the Scala compiler. Combinatorial code-bloat is tackled by specializing for the largest primitive type `long` and by automatically coercing smaller-sized primitives.

State of the Art of Trie Data Structures. Trie data structures were invented 1959 by Briandais [7] and named a year later by Fredkin [12]. An AMT [6, 2] is a trie variant where lookup time is independent from the number of keys stored in the trie. AMTs eliminate empty array slots of nodes by using one bit in a bitmap for each valid outgoing trie branch.

Functional Unordered Collections based on AMTs. A Hash-Array Mapped Trie (HAMT) [3] is a space-efficient trie that encodes the hash code prefixes of elements. HAMTs constitute the basis for purely functional collections that are incrementally constructed and may refer to the unaltered parts of previous states [11, 20]. In previous work we introduced the Compressed Hash-Array Mapped Prefix-tree (CHAMP) [28], a cache-oblivious and canonical HAMT variant that improves the runtime efficiency of iteration (1.3–6.7 x) and equality checking (3–25.4 x) over its predecessor, while at the same time reducing memory footprints.

Functional Lists and Vectors Inspired by HAMTs. Immutable vectors are primarily based on principles of AMTs, because they resulting prefix trees cover densely filled lists. Bagwell and Rompf [4] published a technical report about efficient immutable vectors that improved runtimes of split and merge operations to a logarithmic bound. Stucki et al. [30] improved upon the latter and added a broad scale evaluation.

Concurrent HAMTs. Prokopec et al. [21] worked on mutable concurrent HAMTs that feature iterators with snapshot semantics, which preserve enumeration of all elements that were present when the iterator was created.

3. A Stable Data Type Independent Encoding

Efficient collection data structures on the JVM are typically coded as array-based hashtables. The array core complicates separating commonality from variability to construct a product family. In particular, arrays imply that either all elements are primitives or they are all references. For primitive collections, the absence of a value requires additional encoding (sentinels or bitmaps) to represent **`null`**. AMT-based collections on the other hand do allow fine-grained memory layout choices (per internal node) and are therefore more amenable for encoding a product family of collection data structures. While the API operations and details may differ between variants, we explain how to use the AMT as a fundamental skeleton to support many kinds of efficient immutable collections.

The remainder of this section describes the core concepts of trie-based collections in Feature Description Language (FDL) notation [10]. The full model has been archived [25]. It describes the variability in the domain of collections, making commonalities and differences of configurations explicit, as well as constraints among them.

```
features trie
  EncodingType      : one-of(data, hashOfData)
  EncodingLength    : one-of(bounded, unbounded)
  EncodingDirection : one-of(prefix, postfix)
  ChunkUnit   : one-of(bit, char)
  ChunkLength : int
  DataDensity : one-of(sparse, dense)
  Content     : one-of(mixedNodes, dataAsLeafs)
```

A `trie` is an ordered tree data structure. It is like a Deterministic Finite Automaton (DFA) without any loops, where the transitions are steps of a search path, the internal nodes encode prefix sharing, and the accept nodes hold the stored values. Like with a DFA, a single path represents a single `data` value by concatenating the labels of the edges. An example would be a vector data structure where the index is stored in the path. When we store `hashOfData` however, like in unordered map collections, usually we store a copy at the accept nodes to cater for possible hash collisions. The features `ChunkUnit`, `ChunkLength` and `EncodingDirection` determine the granularity of information encoded by the edges. Encoding direction `prefix` starts at the least-significant bit, whereas `postfix` starts at the most significant bit.

The `trie` model describes the common core characteristics of trie-based collections: each flavor encodes prefixes of either bounded length (e.g., integers) or unbounded length (e.g., strings) with a particular stepping size. Based on any particular `trie` configuration, a code generator can derive the storage and lookup implementation using different (bit-level) operations to split values across the respective paths.

The above describes how the *keys* of a collection are stored in an ordered or unordered collection, but we also cater for more general collections such as maps and relations. To do this we store `Payload` tuples (specification elided) at the accept nodes with variable arity and content. To achieve the required top-level API, a code generator will wrap the internal trie nodes using different visitors to collect the stored data in the required form (e.g., `java.util.Map.Entry`).

The following partial configurations characterize AMTs. First, an AMT-based vector maps from a prefix-encoded index to an element (`index` $\mapsto$ `element`). The `prefix` code direction ensures space efficiency for `dense` vectors, because vector indices usually occupy the least-significant bits:

```
config amt-vector requires EncodingType::data,
  EncodingDirection::prefix, DataDensity::dense
```

Second, a configuration for an unordered hashed collection looks slightly different:

```
config hamt-unordered requires
  EncodingType::hashOfData,
  EncodingLength::bounded, DataDensity::sparse
```

Efficient immutable hash data structures are typically implemented as HAMTs, mapping from `hash(key)` $\mapsto$ `key/value`, in case of a hash-map. In Java, default hash codes are bound in size (32 bit) and assumed to have an almost uniform distribution, so the `EncodingDirection` is not constrained. The size of a hashed collection is usually `sparse`, compared to the 2^{32} space of possible hash codes. The previous two listings describe viable default configurations for vectors and hash-maps of collection libraries. Yet, a feature model allows for customization towards specific workloads (e.g., sparse vectors). For certain efficiency trade-offs it is important to distinguish between HAMT encodings which store `dataAsLeafs` and encodings which allow for `mixedNodes` internally [28].

We currently generate unordered set, map, and multi-map data structures based on the state-of-the-art HAMT variants: HAMT [3], CHAMP [28], and HHAMT [27]. The latter is a generalization of the former two and supports multiple heterogeneous payload categories simultaneously. A subset of the generated collections is distributed with the *capsule* library.[3] In future work we plan supporting vectors and concurrency.

4. Intermediate Generator Abstractions

We use a form of these feature models to configure the Data Structure Code Generator (DSCG) that actually implements each variant.[4] The DSCG is implemented in Rascal, a Domain-Specific Language (DSL) designed for analyzing, processing, transforming and generating source code [17]. We represent variants in trie implementation details using abstract tree grammars with Rascal's **`data`** declarations. In the following section we detail the core intermediate abstractions, necessary to efficiently implement each configuration.

[3] `https://michael.steindorfer.name/projects/capsule/`

[4] `https://michael.steindorfer.name/projects/dscg/`

```
list[Partition] champ_partition_configuration(int bound) = [
  slice("payload", sequence([ generic("K"), generic("V") ]), range(0, bound), forward()),
  slice("node", specific("Node"), range(0, bound), backward()) ];
```

Listing 1. ADT term for the partitioning of a set of family members called CHAMP, parametrized by a size bound (i.e. 32).

```
list[PartitionCopy] applyManipulation(Partition p, Manipulation m:copyAndInsert()) {
  list[PartitionCopy] operations = [ rangeCopy (p, m.beginExpr, m.indexExpr, indexIdentity, indexIdentity),
                                     injection (p, m.indexExpr, valueList = m.valueList),
                                     rangeCopy (p, m.indexExpr, p.lengthExpr, indexIdentity, indexPlus1) ];
  return p.direction == forward() ? operations : reverse(operations);
}
```

Listing 2. Linearization and transformation from domain specific `copyAndInsert` primitive to intermediate abstraction.

Modeling Trie Node Data Layouts and Transformations. The skeleton design is that the out edges of the trie nodes are stored in a array, at least conceptually. Depending on the feature configuration, order, sequence, and types of the elements in the array may differ. For example, these arrays can mix payload and sub-nodes in arbitrary order, or group elements per content category together [28]. We model this variability in array content as follows:

```
data Partition
 = slice (Id,Type,Range,Direction)
 | stripe(Id,Type,Range,Direction,list[Partition]);
```

A partition describes a typed sequence of elements that is limited to a size `Range` (lower and upper bounds). A `slice` is the atomic unit, whereas a `stripe` joins two or more adjacent slices together. The two `Direction` values, `forward` or `backward`, allow advanced slice configurations that —similar to heap and stack— grow from separate fixed bases, to omit the necessity of dynamic partition boundary calculations [28].

Listing 1 shows the partition configuration of a hash-map encoded in CHAMP [28]. CHAMP splits a node's content into two homogeneously typed groups —payload and sub-nodes— that are indexed from different directions. Each partition is delimited in growth (`bound`). Furthermore, a domain specific invariant guarantees space sharing: the sum of sizes of all partitions together must not exceed the `bound`.

DSCG reduces the partition layout to a minimal set of physical arrays, e.g., by grouping adjacent slices of reference types together into a single untyped `stripe`. To reduce memory footprints further, DSCG supports specialization approaches that are specific to AMTs [26, 27].

Synthesizing Linearized Update Operations. DSCG supports twelve primitives for manipulating logical partitions of AMT-based data structures. These primitives cover (lazy) expansion of prefix structures, insert/update/delete on partitions, migration of data between partitions and canonicalization on insert and delete. However, the cost of manipulating data on top of logical partitions increases with added data categories, and furthermore different encoding directions break linearity of copying operations as shown for `copyAndInsert` (in Java):

```
for (int i = 0; i < index; i++)
  dst.setPayload(i, src.getPayload(i));

dst.setPayload(index, new Tuple(key, val));

for (int i = index; i < src.payloadLength(); i++)
  dst.setPayload(i + 1, src.getPayload(i));

for (int i = src.nodeLength(); i >= 0; i--)
  dst.setNode(i, src.getNode(i));
```

If we transform update operations such that they operate on a linearized view of the underlying physical array instead on logical partitions, we can further reduce the number of back-end generator primitives to two —`rangeCopy` that supports index shifts, and `injection` of payload— as shown in Listing 2. A linearized view effectively turns copy operations into stream processing operations, where the source and destination arrays are traversed with monotonous growing indices front to back. Adjacent `rangeCopy` operations can be fused together to increase efficiency as shown below (in Java):

```
offset += rangeCopy (src, dst, offset, index);
delta  += injection (dst, offset, key, val);
offset += rangeCopy (src, offset, dst, offset +
    delta, length - index);
```

5. Conclusion

The Array Mapped Tries skeleton is a common framework for generating fast immutable collection data structures. Our feature model covers both variants that occur in the wild, and supports novel heterogeneous variants [27]. The generated code is efficient, overall outperforming competitive state-of-the-art collections [28, 27], and —when specialized for primitive data types— they match the memory footprints of best-of-breed primitive collections [27].

Based on this evidence of the efficacy of the feature model and the intermediate abstractions for DSCG, we will extend it further to generate a complete Software Product Line of trie-based immutable collections.

References

[1] V. Alves, D. Schneider, M. Becker, N. Bencomo, and P. Grace. "Comparitive Study of Variability Management in Software Product Lines and Runtime Adaptable Systems". In: *Proceedings of VaMoS '09*. Universität Duisburg-Essen, 2009.

[2] P. Bagwell. *Fast And Space Efficient Trie Searches*. Tech. rep. LAMP-REPORT-2000-001. Ecole polytechnique fédérale de Lausanne, 2000.

[3] P. Bagwell. *Ideal Hash Trees*. Tech. rep. LAMP-REPORT-2001-001. Ecole polytechnique fédérale de Lausanne, 2001.

[4] P. Bagwell and T. Rompf. *RRB-Trees: Efficient Immutable Vectors*. Tech. rep. EPFL-REPORT-169879. Ecole polytechnique fédérale de Lausanne, 2011.

[5] T. J. Biggerstaff. "A Perspective of Generative Reuse". In: *Annals of Software Engineering* 5.1 (1998).

[6] R. Bird. "Two Dimensional Pattern Matching". In: *Information Processing Letters* 6.5 (1977).

[7] R. de la Briandais. "File Searching Using Variable Length Keys". In: *Proceedings of IRE-AIEE-ACM '59 (Western)*. ACM, 1959.

[8] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2001.

[9] K. Czarnecki and U. W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. ACM Press, 2000.

[10] A. van Deursen and P. Klint. "Domain-Specific Language Design Requires Feature Descriptions". In: *Journal of Computing and Information Technology* 10.1 (2002).

[11] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan. "Making Data Structures Persistent". In: *Proceedings of STOC '86*. ACM, 1986.

[12] E. Fredkin. "Trie Memory". In: *Communications of the ACM* 3.9 (1960).

[13] S. Hallsteinsen, M. Hinchey, S. Park, and K. Schmid. "Dynamic Software Product Lines". In: *Computer* 41.4 (2008).

[14] P. Hawkins, A. Aiken, K. Fisher, M. Rinard, and M. Sagiv. "Concurrent Data Representation Synthesis". In: *Proceedings of PLDI '12*. ACM, 2012.

[15] P. Hawkins, A. Aiken, K. Fisher, M. Rinard, and M. Sagiv. "Data Representation Synthesis". In: *Proceedings of PLDI '11*. ACM, 2011.

[16] P. Helland. "Immutability Changes Everything". In: *Communications of the ACM* 59.1 (2015).

[17] P. Klint, T. van der Storm, and J. Vinju. "Rascal: A Domain Specific Language for Source Code Analysis and Manipulation". In: *Proceedings of SCAM '09*. IEEE, 2009.

[18] C. Loncaric, E. Torlak, and M. D. Ernst. "Fast Synthesis of Fast Collections". In: *Proceedings of PLDI '16*. ACM, 2016.

[19] D. McIlroy. "Mass-Produced Software Components". In: *Proceedings of NATO Software Engineering Conference*. Scientific Affairs Division, NATO, 1969.

[20] C. Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1999.

[21] A. Prokopec, N. G. Bronson, P. Bagwell, and M. Odersky. "Concurrent Tries with Efficient Non-blocking Snapshots". In: *Proc. of PPoPP '12*. ACM, 2012.

[22] E. Schonberg, J. T. Schwartz, and M. Sharir. "Automatic Data Structure Selection in SETL". In: *Proceedings of POPL '79*. ACM, 1979.

[23] O. Shacham, M. Vechev, and E. Yahav. "Chameleon: Adaptive Selection of Collections". In: *Proceedings of PLDI '09*. ACM, 2009.

[24] Y. Smaragdakis and D. Batory. "DiSTiL: A Transformation Library for Data Structures". In: *Proceedings of DSL'97*. USENIX Association, 1997.

[25] M. J. Steindorfer. *Towards a Feature Model of Trie-Based Collections*. 2016. DOI: 10.5281/zenodo.59739.

[26] M. J. Steindorfer and J. J. Vinju. "Code Specialization for Memory Efficient Hash Tries (Short Paper)". In: *Proceedings of GPCE '14*. ACM, 2014.

[27] M. J. Steindorfer and J. J. Vinju. "Fast and Lean Immutable Multi-Maps on the JVM based on Heterogeneous Hash-Array Mapped Tries". In: *ArXiv e-prints* (2016). arXiv: 1608.01036 [cs.DS].

[28] M. J. Steindorfer and J. J. Vinju. "Optimizing Hash-array Mapped Tries for Fast and Lean Immutable JVM Collections". In: *Proceedings of OOPSLA '15*. ACM, 2015.

[29] M. J. Steindorfer and J. J. Vinju. "Performance Modeling of Maximal Sharing". In: *Proceedings of ICPE '16*. ACM, 2016.

[30] N. Stucki, T. Rompf, V. Ureche, and P. Bagwell. "RRB Vector: A Practical General Purpose Immutable Sequence". In: *Proceedings of ICFP '15*. ACM, 2015.

[31] V. Ureche, C. Talau, and M. Odersky. "Miniboxing: Improving the Speed to Code Size Tradeoff in Parametric Polymorphism Translations". In: *Proceedings of OOPSLA '13*. ACM, 2013.