To-Many or To-One? All-in-One! Efficient Purely Functional Multi-maps with Type-Heterogeneous Hash-Tries

Michael J. Steindorfer
Delft University of Technology, The Netherlands
m.j.steindorfer@tudelft.nl

Jurgen J. Vinju
Centrum Wiskunde & Informatica, The Netherlands
TU Eindhoven, The Netherlands
jurgen.vinju@cwi.nl

Abstract

An immutable multi-map is a many-to-many map data structure with expected fast insert and lookup operations. This data structure is used for applications processing graphs or many-to-many relations as applied in compilers, runtimes of programming languages, or in static analysis of object-oriented systems. Collection data structures are assumed to carefully balance execution time of operations with memory consumption characteristics and need to scale gracefully from a few elements to multiple gigabytes at least. When processing larger in-memory data sets the overhead of the data structure encoding itself becomes a memory usage bottleneck, dominating the overall performance.

In this paper we propose AXIOM, a novel hash-trie data structure that allows for a highly efficient and type-safe multi-map encoding by distinguishing inlined values of singleton sets from nested sets of multi-mappings. AXIOM strictly generalizes over previous hash-trie data structures by supporting the processing of fine-grained type-heterogeneous content on the implementation level (while API and language support for type-heterogeneity are not scope of this paper). We detail the design and optimizations of AXIOM and further compare it against state-of-the-art immutable maps and multi-maps in Java, Scala and Clojure. We isolate key differences using microbenchmarks and validate the resulting conclusions on a case study in static analysis. AXIOM reduces the key-value storage overhead by 1.87 x; with specializing and inlining across collection boundaries it improves by 5.1 x.

CCS Concepts • **Theory of computation → Data structures design and analysis**; • **Information systems →** *Point lookups*; Data compression; Hashed file organization; Indexed file organization;

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
PLDI'18, June 18–22, 2018, Philadelphia, PA, USA
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5698-5/18/06…$15.00
https://doi.org/10.1145/3192366.3192420

Keywords Data structures, persistent data structures, functional programming, hashtable, multi-map, many-to-many relation, graph, optimization, performance, JVM.

ACM Reference Format:
Michael J. Steindorfer and Jurgen J. Vinju. 2018. To-Many or To-One? All-in-One! Efficient Purely Functional Multi-maps with Type-Heterogeneous Hash-Tries. In *Proceedings of 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'18).* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3192366.3192420

1 Introduction

Purely functional data structures [17] have their origins in the domain of functional programming, but are nowadays available in many widely-spread programming languages, such as Java, Scala, Clojure, Erlang, Haskell and F#. From a user's perspective, purely functional data structures are beneficial in many ways: immutability for collections implies referential transparency without giving up on sharing data, it satisfies safety requirements for having co-variant subtypes [14], and it guarantees safe sharing of data structure instances in the presence of concurrent computations.

This paper addresses the challenges of optimizing purely functional multi-maps for standard libraries of programming languages. A multi-map is a data structure that acts as an associative array storing possibly multiple values with a specific key. Typically multi-maps are used to store graphs or many-to-many relations, which occur naturally in application areas such as compilers, runtimes of programming languages, or static analysis of object-oriented software. In some applications it is the case that the initial raw data is many-to-one, and further processing or exploration incrementally leads to a many-to-many mapping for some of the entries. In other applications the distribution of sizes of the range sets in the raw data is highly skewed, such as when representing program dependence graphs [12]. The number of values associated with a specific key is then practically always very low, yet there are possibly numerous exceptions to cater for nevertheless, where many values end up being associated with the same key. A key insight in the current paper is that we can exploit highly common skewed distributions to save memory for the most frequent cases.

In line with recent efforts of optimizing generic general purpose collections [10, 19, 24, 26], we aim to improve collection data structures towards memory-intensive applications, which is a preliminary for processing larger data sets that fit into main memory.

On Java Virtual Machine (JVM) languages such as Java, Scala and Clojure, relations are not language-supported; rather the standard libraries of the aforementioned languages allow the construction of multi-maps by using sets as the values of a normal polymorphic map. The goal of this paper is to overcome the limitations of these existing implementations of multi-maps on the JVM, improving drastically on the memory footprint without any other loss of efficiency. While comparable multi-maps come with a mode of 65.37 B overhead per stored key/value item, the most compressed encoding in this paper reaches an optimum of 12.82 B.

Contributions

1. We contribute AXIOM, a novel hash-trie data structure that is a template for implementing data structures that require storage and retrieval of fine-grained type-heterogeneous content.[1]
2. We detail a multi-map implementation based on AXIOM that improves overall run-time efficiency and reduces memory footprints by 1.87 x–5.1 x over idiomatic multi-maps in Clojure and Scala.
3. We show that AXIOM strictly generalizes over the state-of-the-art hash-trie data structures, namely HAMT [2] and CHAMP [24], which are special instances of AXIOM.

All source code of data structures and benchmarks discussed in this paper is available online.[2]

2 A Primer on Hash-Trie Data Structures

Hash-trie data structures form the basis of efficient immutable unordered hash-set and hash-map implementations that are contained in standard libraries of programming languages such as Clojure and Scala [2, 24]. A general trie [5, 7] is an n-ary search tree that features a large branching factor. In a hash-trie, the search keys are the bits of the hash codes of the elements that are stored in the prefix-tree structure. Hash-tries are by construction memory efficient due to prefix sharing, but also because child nodes are allocated lazily only if the prefixes of two or more elements overlap.

While hash-tries allow implementing both mutable and immutable data types, they are the de-facto standard for efficient immutable collections, whereas array-based hashtables keep predominating mutable collections. The main difference between mutable and immutable hash-trie variants lies in *how* and *when* trie nodes are reallocated. Mutable hash-tries reallocate a node only if the node's arity changes [2], otherwise nodes are updated in-place. In contrast, immutable hash-tries perform path-copying [17, 20] by reallocating a whole branch, i.e., the node to-be-updated and all its parent nodes. The resulting trie satisfies the immutability property without modifying the previous version: it consists of a new root node and an updated branch, while structurally sharing all unmodified branches between both instances.

Such immutable data structures that are incrementally constructed and that structurally share common data are called *persistent data structures* [6, 11, 17]. Note that the term persistency in the context of immutable data structures has nothing in common with object serialization or the likewise named property of database systems. Overall, persistency is a much stronger property than immutability. While immutability prohibits mutation, persistency enables efficient derivations of new data structure instances. Compared to naïve copy-on-write data structures, only a small logarithmic delta of a data structure's object graph is path-copied.[3]

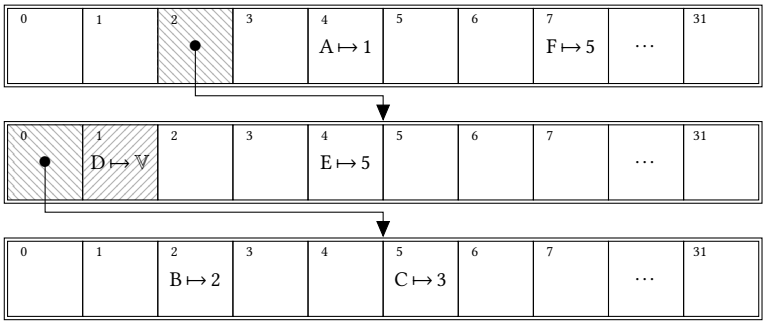
Recapitulating Array Mapped Tries. In order to discuss the contributions of the AXIOM data structure, we first concisely recapitulate how Array Mapped Tries (AMTs) in general work, and how in particular Hash-Array Mapped Tries (HAMTs) work. Figure 1 illustrates an uncompressed AMT multi-map data structure that stores mappings from objects to integers, i.e., the tuples $A \mapsto 1$, $B \mapsto 2$, $C \mapsto 3$, $D \mapsto 4$, $D \mapsto -4$, and $F \mapsto 5$. Note that the key D maps to multiple numbers. For the moment, we ignore how multi-mappings are internally stored and simply denote that $D \mapsto \mathbb{V}$, where $\mathbb{V}$ is an arbitrary set of values. Figure 1b lists the hash codes of tuple keys in base 10 and base 32. The hashes determine the prefix-tree structure of the example in Figure 1a. Each digit in the base 32 hash code notation encodes a distinct 5-bit chunk of a key's hash code to which we will refer to as *mask*. E.g., the first digit masks bits 0–4 bits of the hash code, the second digit masks to the bits 5–9, etc. The masks are then subsequently used as indices to navigate through the prefix tree. As the name suggests, in an AMT each node is implemented as an array of fixed size, e.g., in our case the array is of size 32 to cater for all possible 5-bit mask values. In Figure 1a, the array indices are displayed in the left upper corner of each cell, the empty cells 8–30 are elided in favor of a concise graphical representation. For immutable collections, 5-bit prefixes experimentally yield a good performance balance between search and update operations [3].

Lookup. Search is a recursive operation, which navigates through the trie by means of indexing. We start by indexing

[1] This paper mainly describes the machinery of processing type-heterogeneous data with hash-trie data structures. API and language support for type-heterogeneity are not explicitly discussed in this paper, since the AXIOM multi-map implementation leverages type-heterogeneity only internally for performance optimizations without exposing it to the API user.

[2] https://michael.steindorfer.name/papers/pldi18-artifact

[3] On the JVM, many popular third-party collection libraries —including Google's *Guava* library— solely contain immutable data structures, but not persistent data structures. Updating immutable data structures by means of copy-on-write is orders of magnitude slower than updating persistent data structures that are, for example, contained in Clojure or Scala.



(a) uncompressed prefix search tree where each node is an array

$hash(A) = 4_{10} = 4\,0\,0\ldots_{32}$
$hash(B) = 2050_{10} = 2\,0\,2\ldots_{32}$
$hash(C) = 5122_{10} = 2\,0\,5\ldots_{32}$
$hash(D) = 34_{10} = 2\,1\,0\ldots_{32}$
$hash(E) = 130_{10} = 2\,4\,0\ldots_{32}$
$hash(F) = 7_{10} = 7\,0\,0\ldots_{32}$

(b) hash codes of keys in base 32

Figure 1. Example of an uncompressed AMT data structure storing multi-map tuples in (a). The hash codes of the tuple keys are listed under (b) in base 10 and base 32. The base 32 digits are used to navigate through the tree of fixed-size arrays. The first digit is used to index the array on level 1, the second digit is the index for array on level 2, etc.

into the root node's array with the first mask value. If the array slot is empty, no search key is associated with the prefix denoted by mask. If the array slot is indeed occupied, we have to distinguish two further cases: it contains a key/value or key/value-set pair, or it points to a nested array. In the former case we can compare the search keys to decide if the search key is present. In the latter case, we need to recurse with the remaining mask values. E.g., to lookup key A, we index into the root node with mask 4 and successfully encounter a key/value pair with its key matching our search key. For looking up key C, we have to recurse twice before we can distinguish it from object B in the leaf node with mask 5. Note that values B and C share the prefix sequence 2 and 0.

Insertion. Insertion reuses the navigation from lookup. The element to be inserted is placed in the trie as soon as it can be unambiguously distinguished from all other elements by its prefix. If necessary, insertion lazily expands the trie and adds new sub-nodes to distinguish prefixes on the next level.

Deletion. This operation also reuses the navigation from lookup. After a search key is located in the tree, the payload is removed from the node where it was stored. Deletion may yield suboptimal tries. E.g., in Figure 1, removing the tuple with key C leaves the tuple with key B as only element in the node left, while it could be more efficiently stored one level higher. Keeping a trie minimal and canonical enables significant performance improvements [24].

Sparsity. Tries can encode arbitrary search keys that are binary comparable. A HAMT is a specific instance of an AMT that encodes the hash codes of the search keys in the prefix-tree structure. As such, a hash-trie needs to support resolving of hash collisions in case the prefixes are identical but the search keys differ. For hash-tries, a typical uniform distribution of hash values always yields a sparse AMT with a majority of empty cells per node. In the context of collections, such a in-memory representation would be very inefficient. E.g., Figure 1 requires three arrays with in total 96 cells for storing references to seven key/value pairs. For better memory efficiency hash-tries normally apply some form of compression. The most simple and wide-spread approach is to compact the sparse array with the help of a single 32-bit bitmap [3], eliminating all empty cells. This compaction comes at a cost: it requires dynamic type checks to recover which type of content is actually stored in an array cell, and it implies that all content shares the same structural representation. A recent approach [24] removes the cost of dynamic type checks using one additional bitmap to distinguish stored payload from sub-trees, but conceptually does not generalize to multiple type-heterogeneous content categories.

3 The AXIOM Prefix-Search Tree

The design goal of AXIOM is to efficiently store and discriminate on fine-grained type-heterogeneous content. This is of importance, because many data structure design challenges can be mapped to heterogeneous optimization problems.

In the case of a new immutable multi-map data-structure, mappings of any cardinality need to be supported within a single data structure: be it $1:1$, $n:1$, or $n:n$. We want to let memory performance improve or degrade gracefully as the arities of the domain and range of a relation incrementally grow or shrink during a computation, e.g., during constraint solving. The unavoidable run-time overhead we introduce, to be able to compress and switch on various content representations, must be kept small for achieving a balanced collection design that yet significantly saves memory. We will evaluate the result experimentally in Sections 4 and 5.

Although the original and elegant hash-trie encoding by Bagwell [2] forms the basis of our design, it is itself not amenable directly to fine-grained type-heterogeneous representations, such as needed for optimizing multi-map data

structures. Instead we need a different representation. Similar to the changes made for more recent immutable sequence data structures [19, 26] and binary search trees [21] we need to break out of the original hash-trie design and add some initial overhead to achieve what we want: a leaner and faster implementation of multi-maps and likewise data structure designs (that require fine-grained type-heterogeneity).

Foundations of AXIOM. The first concept behind the AXIOM prefix-tree data structure is to use a single array to store many different types of content. This allows us to specialize not only for a sparse domain like a HAMT, but also for different classes of cardinality for the range of a relation: singletons can be inlined, small collections can be specialized and large collections can be used as a general fallback. Moreover, content can migrate easily from one to another content representation, without having to frequently reallocate the underlying array. As a result, the expected overhead per stored element should drop significantly.

The second concept behind AXIOM is the grouping of similar content together, in consecutive array slices. For a multi-map this translates to grouping per trie node all $1:1$ tuples, all $1:n$ tuples, and all sub-nodes in distinct array slices together, instead of arbitrarily mixing them. This grouping then enables an efficient bitmap encoding to help distinguish which array element has which type without resorting to dynamic type checks at runtime.

The third concept behind AXIOM is that individual elements do not need to be checked for their specific type, during iteration, streaming, or when batch-processing elements. We argue that this is the main reason why the new encoding for separating multiple content representations performs well with iteration-based algorithms: in AXIOM internal type-heterogeneity comes at a low cost. Observations from implementing speculative runtime optimizations for collection processing in dynamic languages indeed suggest that avoiding individual type checks is a key enabler for obtaining good overall performance [4].

3.1 Generalizing Existing Hash-Trie Data Structures

The two contenders for implementing efficient unordered hashtables for collection libraries are the HAMT [2] and the CHAMP [24] data structure. The principles of these data structures were already covered in Section 2, with exception of how they deal with sparsity and compression. In the following, we first detail how HAMT and CHAMP encode the three minimal states of a hash-trie —`EMPTY`, `PAYLOAD`, and `NODE`— and, second, how AXIOM efficiently generalizes these two data structures by supporting multiple (type-heterogeneous) payload categories.

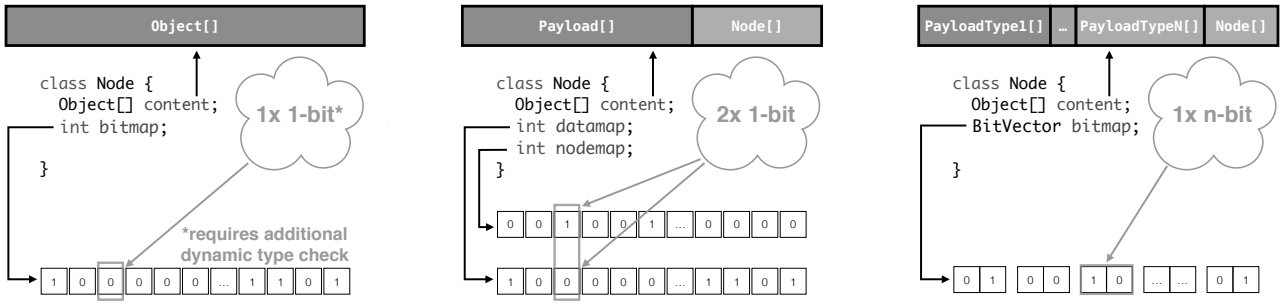
A Note on Type-Safety. In the context of hash-tries we interchangeably use the term *type-safety* for type-casts that are guaranteed to succeed, i.e., that are never type-violating. This is an important detail to keep in mind, since generic implementations of all three data structures on the JVM use arrays of type `Object[]` for storing the payload in the search tree, regardless of the generic type parameters. Yet, type-safety is guaranteed by either using casts after **`instanceof`** checks, or checked casts that rely on explicit type meta-data that is efficiently stored in bitmaps. Figure 2 visualizes the conceptual differences between HAMT, CHAMP and AXIOM:

HAMT (cf. Figure 2a) uses a 1-bit per branch encoding that is insufficient to encode a HAMT's three possible branch states (i.e., `EMPTY`, `PAYLOAD`, and `NODE`). The single bitmap serves to identify if a branch is occupied or empty, to then allocate a dense array (`Object[]`) without empty cells. The resulting dense array contains an untyped mix of `PAYLOAD` and `NODE` instances. To discriminate between both cases, a HAMT relies on type checks at run-time, e.g., **`instanceof`** on the JVM.

CHAMP (cf. Figure 2b) is a successor design of HAMT that explicitly encodes all three states —`EMPTY`, `PAYLOAD`, and `NODE`— with bitmaps and does not require type checks at run-time. More specifically, CHAMP uses two distinct bitmaps: one each for identifying the `PAYLOAD` and `NODE` states, while the `EMPTY` state is implied by not being present in either bitmap. Due to the explicit bitmap encoding of all three states, CHAMP can permute the content of the untyped array —grouping together all payload and all nodes— while still being able to track the hash-prefixes to the corresponding cells in the dense, compacted, and permuted array. The permutation turned out to be the key design element relevant to increase cache locality, decrease memory footprints and to drastically improve the performance of iteration (on average 1.3–6.7 x) and equality checking (on average by 3–25.4 x).

Nevertheless, CHAMP also has limitations that prohibit extending it to a generalized type-heterogeneous data structure. At lookup, insertion, and deletion, CHAMP sequentially checks (in order) if the prefix is contained in the `PAYLOAD` bitmap, or if it is contained in the `NODE` bitmap. Otherwise, the prefix implicitly belongs to the `EMPTY` category. Explicitly storing membership of each group (i.e., `PAYLOAD` and `NODE`) in a distinct bitmap, but not explicitly storing `EMPTY` incurs root limitations when extrapolating CHAMP's design to type-heterogeneity, as illustrated in Listing 1. Firstly, memory overhead increases by adding dedicated bitmap for each group membership to test (e.g., in the case of multi-maps requiring one additional bitmap for the $1:n$ payload category). Secondly, negative lookups become slow, because the `EMPTY` state is not explicitly encoded. Thirdly, offset-based indexing is tedious and expensive, because offsets must be aggregated by counting bits over scattered bitmaps.

AXIOM (cf. Figure 2c) aims to inherit the beneficial performance characteristics of CHAMP by relying on content permutation, while eliminating the restrictions that made the data structure only applicable to (homogeneous) hash-set and hash-map implementations. We first discuss, bitmap



(a) HAMT **(b)** CHAMP **(c)** AXIOM

Figure 2. Visualization of the conceptual encoding differences between HAMT, CHAMP and AXIOM.

```
void template(int keyHash, int shift) {
  int mask = mask(keyHash, shift);
  int marker = 1 << mask;

  if ((datamap1() & marker) != 0) {
    int index = index(datamap1(), marker);
    ...code for lookup, insert or delete ...
  } else if ((datamapN() & marker) != 0) {
    int index = count(datamap1())
              + count(...)
              + index(datamapN(), marker);
    ...code for lookup, insert or delete ...
  } else if ((nodemap() & marker) != 0) {
    int index = count(datamap1())
              + count(...)
              + count(datamapN(), marker);
              + index(nodemap(), marker);
    ...code for lookup, insert or delete ...
  } else {
    // default: empty (marker not found in any bitmap)
    ...code for lookup, insert or delete ...
  }
}
```

Listing 1. Extrapolated CHAMP template using linear scanning for dispatching on type-heterogeneous payload.

```
void template(long bitmap, int keyHash, int shift) {
  int mask = mask(keyHash, shift);
  int type = (int) ((bitmap >>> (mask << 1)) & nBitMask);

  int relativeIndex = index(bitmap, type, mask);
  int absoluteIndex = offset(type, histogram(bitmap))
                    + weight(type) * relativeIndex;

  switch (type) {
  case EMPTY:
    ...code for lookup, insert or delete ...
    break;
  case PAYLOAD_CATEGORY_1:
    ...code for lookup, insert or delete ...
    break;
  case PAYLOAD_CATEGORY_2:
    ...code for lookup, insert or delete ...
    break;
  case NODE:
    ...code for lookup, insert or delete ...
    break;
  }
}
```

Listing 2. AXIOM template for lookup, insertion, and deletion that processes a `long` bitmap with 2-bit wide entries.

representations, before discussing the necessary abstractions required to enable efficient content permutations. In contrast to CHAMP, AXIOM uses a single bit-vector multi-bit type-tags:

Tagging Type-Heterogeneous Payload: We assign each category that we want to discriminate a unique integer constant. E.g., a multi-map implemented with AXIOM is supposed to discriminate between two different content representations: `PAYLOAD_CATEGORY_1 = 1` identifies a key/value pair with an inlined singleton value, and `PAYLOAD_CATEGORY_2 = 2` hints that a key is associated with a reference to a nested set of multiple values. Sequential enumeration naturally extends to an arbitrary number of categories.

Tagging a Trie's Internal Representations: Next to the payload categories, we have to mark nested sub-trees with a distinct `NODE` category. By convention we assign the highest category number to sub-trees, i.e., `NODE = 3` in the case of an AXIOM multi-map. AXIOM accounts for sparsity that arises from hashing with an extra tag: by convention we decided to use the lowest number: `EMPTY = 0`. These choices allows subsequent optimizations, as we will detail later in this section.

Note that treating type-heterogeneous payload tags and internal representations alike, makes every category membership explicit. Further, it allows generalizing over HAMT and CHAMP. The former case uses only tag `EMPTY` with subsequent

dynamic type-checks, whereas the latter uses the categories `EMPTY`, `PAYLOAD`, and `NODE`. The general type-heterogeneous case utilizes a variable amount of payload categories that needs to be fixed statically at design time of the data structure.[4] We refer to AXIOM's bounded type-heterogeneity as type-heterogeneity of rank k, where k denotes the maximum number of supported heterogeneous types the data structure can discriminate. For any k, a AXIOM node requires $n = \lceil log_2(k + 2) \rceil$ bits per branch. The constant 2 in the formula caters for the obligatory `EMPTY` and `NODE` cases.

As highlighted in Listing 2, the unified tags allow fast dispatching with **`switch`** statements instead of linear probing, without being limited by a larger number of case distinctions. Note that all case blocks access the (payload) data by indexing into the `Object[] content` array with the `absoluteIndex`. Since each case block corresponds to a specific type that was recovered from the bitmap meta-data, AXIOM can safely perform the corresponding type casts in each branch, which are guaranteed to succeed.

Running Example: Multi-Maps. We continue detailing the AXIOM's concepts that generalize towards k data categories, while applying it to an optimized type-heterogeneous multi-map that distinguishes between $1:1$ and $1:n$ tuples.[5]

In the case of a multi-map, AXIOM discriminates in total between four states. A direct binary encoding of those four states requires two bits ($n = 2$). We map one 2-bit variable to each branch in our trie. With a branching factor of 32, we require 32 2-bit variables to encode each branch's state. On the JVM, an 8-byte primitive value of type **`long`** suffices to concisely store all 32 2-bit variables consecutively in a single bitmap. In the resulting bitmap, the first two bits designate the state of the first branch, while the second two bits designate the state of the second branch, etc.

3.2 Abstractions for Scalable Type-Heterogeneity

AXIOM's regrouping into different slices is achieved by permuting the content of the array cells, while at the same time keeping track of the `mask` values that are associated with the elements. Permutations occur each time a value is inserted, removed, or converts its internal representation, e.g., in the case of a multi-map converting a singleton to a multi-mapping upon insertion, converting from a payload to a sub-node when prefixes collide, or inlining a node's single payload value in its parent node upon compaction.

Figure 3 illustrates the step-by-step construction of an AXIOM multi-map that is equivalent to the trie of Figure 1a. In the sequence of incremental insertions, two changes of group membership happen that require permutation. First, when comparing Figures 3a and 3b we can observe that $A \mapsto 1$ swaps place with a newly extended sub-tree at the root node tuple. Second, when comparing Figures 3c and 3d we can observe that the insertion of $D \mapsto -4$, triggers a conversion from an key/value pair to a key with a nested set of values.

AXIOM needs to fulfill the following two criteria in order to enable an efficient implementation of permutation. First, calculating any content group's offset within the node's array needs to be supported efficiently. Second, relative indexing into an individual group needs to be supported efficiently and be aware of compressions. We will address the two aforementioned challenges in Sections 3.3 and 3.4 respectively.

3.3 Content Histograms

Streaming or iterating over a AXIOM's content requires information about the amount and types of the whole content that is stored in a trie node. Studies on homogeneous and heterogeneous data structures [4] have shown that avoiding checks on a per elements basis is indeed relevant for achieving good performance. In order to avoid such checks while indexing into the shared array, we use histograms:

```
int[] histogram = new int[1 << n];
for (int branch = 0; branch < 32; branch++) {
  histogram[(int) bitmap & nBitMask]++;
  bitmap = bitmap >>> n;
}
```

This listing abstracts over the number of type distinctions and executes in 32 iterations. The constant n is set to 2 due to our 2-bit patterns, and consequently `nBitMask` is `0b11` — it has the lowest n bits set to 1. In its generic form, the code profits from default compiler optimizations such as scalar replacement [22], to avoid allocations on the heap, and loop unrolling. Note that for a fixed n partial evaluation can remove the intermediate histograms completely.

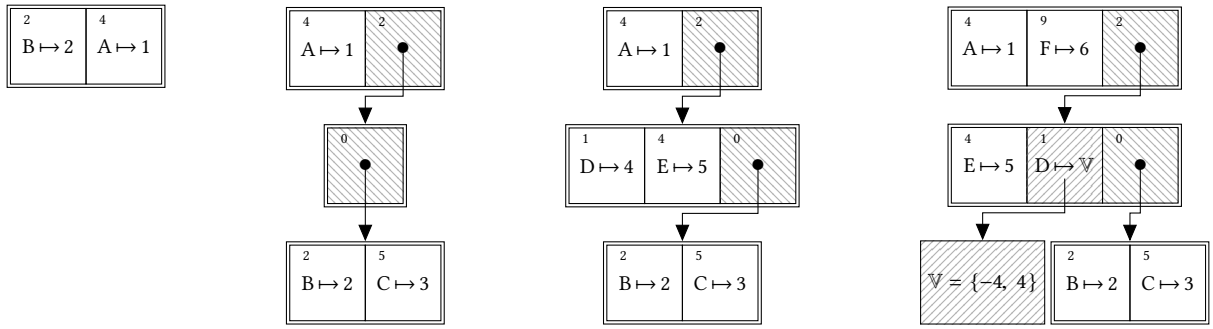
For streaming, iteration or batch-processing of data, the histograms avoids re-counting the elements of individual categories at run-time. And, the otherwise complex code for trie-node iteration reduces to looping through the two-dimensional histogram using two integer indices.

An added benefit is that inlined values although stored out of order, will be iterated over in concert, avoiding spurious recursive traversal and its associated cache misses [24]. Finally, to avoid iterating over empty positions in the tail, iteration can exit early when the running counter reaches its upper bound: with histograms, the total number of elements, regardless of their types, can be calculated with the formula `32 - histogram[EMPTY] - histogram[NODE]`.

Calculating Groups Lengths and Offsets. A histogram allows to efficiently derive length and offset properties. For a type-heterogeneous AXIOM multi-map data structure, we can provide a vector of *weights* $= [0, 2, 2, 1]$ that defines the amount object references a group-specific entry requires.

[4] AXIOM therefore differs from run-time type-heterogeneous data structures structure that often are difficult to optimize.

[5] We assume that AXIOM will be used a template to design arbitrary collection data structures that require fine-grained type-heterogeneity.



(a) $A \mapsto 1, B \mapsto 2$ **(b)** $C \mapsto 3$ **(c)** $D \mapsto 4, E \mapsto 5$ **(d)** $D \mapsto -4, F \mapsto 6$.

Figure 3. An incrementally constructed and compressed AXIOM prefix search tree data structure: empty array slots are eliminated and non-empty slots are permuted such that same-typed elements are juxtaposed. In this example nodes contains first a sequence of $1:1$ mappings, followed by $1:n$ mappings, and at the end a sequence of sub-trees. $1:1$ mappings are stored in-place, whereas $1:n$ mapping allocate and nest a set data structure.

`EMPTY` cells are dropped by compression ($weight = 0$), whereas both `PAYLOAD_CATEGORY_1` and `PAYLOAD_CATEGORY_2` store two references ($weight = 2$): the former case refers to an inlined key/value pair, the latter case refers to a key that is accompanied by a reference to a nested set. The category `NODE` stores a single sub-node reference ($weight = 1$). With the histogram and the weight vector, we can now calculate the length properties of the groups by component-wise multiplication: $length_i = histogram_i * weight_i$. From the *length* vector we infer the offsets, and thus boundaries, of the groups that are all stored within a single array:

$$offset_i = \begin{cases} 0 & \text{if } i = 0 \\ offset_{i-1} + length_{i-1} & \text{if } i \neq 0 \end{cases}$$

Histogram, length and offset vectors provide all information necessary for batch-processing AXIOM trie-nodes, but also identify group offsets that are required for relative indexing into a group. The operation of relative indexing, on basis of AXIOM's bitmap encoding, is discussed next.

3.4 Relative Indexing into a Particular Group

While some operations require knowledge about the distribution of all content stored in a node, others are satisfied by accessing a single element of one particular data category. In an uncompressed AMT the `mask` directly reflects the array index. In an AMT the node's array is therefore totally ordered by the monotonously increasing `mask` values. This property can be exploited for relative indexing into a compressed content group [2]. In AXIOM, `mask` and index do usually differ due to permutation of the cells. AXIOM preserves the total ordering amongst values that belong to a specific group according to their mask values in presence of permutation. The concept of relative indexing [2] therefore remains applicable to individual groups, however hardware and standard library bit-counting Application Program Interfaces (APIs), which are also used in case of HAMT, do not support counting occurrences of n-bit long literals in bit-vectors efficiently; a challenge that we address in this section.

```
1  int index(long unfilteredBitmap, int type, int mask) {
2    long bitmap = filter(unfilteredBitmap, type);
3    long marker = 1L << (mask << 1);
4    return Long.bitCount(bitmap & (marker - 1));
5  }
6
7  long filter(long bitmap, int type) {
8    long mask = 0x5555555555555555L;
9    long masked0 = mask & bitmap;
10   long masked1 = mask & (bitmap >> 1);
11
12   switch (type) {
13   case EMPTY:
14     return (masked0 ^ mask) & (masked1 ^ mask);
15   case PAYLOAD_CATEGORY_1:
16     return masked0 & (masked1 ^ mask);
17   case PAYLOAD_CATEGORY_2:
18     return masked1 & (masked0 ^ mask);
19   case NODE:
20     return masked0 & masked1;
21   }
22 }
```

Listing 3. Reducing of multi-bit patterns to single bits for relative indexing into a group's sorted sequence.

Template for Lookup, Insertion, and Deletion. Listing 2 illustrates a Java source template for how lookup, insertion, or deletion operations would retrieve a cell's group membership. The template function takes three arguments: the bitmap itself, the hash code of the search key, and the `shift`

value that corresponds to the trie-node's level and subsequently allows retrieving the 5-bit `mask` of the `keyHash` according to the node's level.

The `mask` (line 2) is used to recover the `type` pattern (line 3) from the bitmap: first, the offset of the `type` pattern corresponding to `mask` value is calculated with `mask << 1`; second, the 2-bit pattern is shifted to the start of the bitmap with the help of the offset, and finally retrieved with the `nBitMask`.

With a `type`, the lookup, insertion or deletion operation then directly jumps to the relevant case handler (lines 9–22). The search key's relative index within a group (line 5) is calculated by counting how many times the same `type` occurs in the bitmap before the search key's position. Remember that all elements within a group remain totally-ordered according to their `mask` values. The `relativeIndex`, together with the vectors that are derived from the histogram (`offset` and `weight`), lets AXIOM determine the absolute index (line 6) of the content within the node's array. The weight is used for a group-internal offset, because each content category of can be mapped to consume multiple physical array slots. We will continue with first discussing preliminary performance considerations of relative indexing on platforms such as the JVM, before detailing the algorithm.

Performance Considerations. Bit-counting APIs that are efficiently compiled to hardware instructions on the `X86` / `X86_64` architectures are frequently encountered in standard libraries of programming languages. E.g., the Java standard library does contain bit count operations for **`int`** and **`long`**, which count the number of bits that are set to 1. The aforementioned APIs unfortunately do not cover counting n-bit patterns with $n > 1$, which are necessary for our encoding. In order to leverage the efficiency of the hardware bit-count operations, we introduce bitmap pre-processing filters that simplify multi-bit patterns to single bits. Listing 3 (lines 7–22) shows the filter function, which receives two arguments: an unfiltered bitmap, and the bit-pattern of the `type` to search for. Matching entries are reduced to 01 (i.e., a single bit set to 1), while all non-matching entries are reset to 00. The resulting bitmap can be fed into standard bit-counting operations.

Implementation Details. Listing 3 (lines 1–5) shows a complete Java implementation of the relative indexing operation. For ease of understanding, we will explain the operation by example, by calculating the index of the tuple $F \mapsto 6$ from the root node from Figure 3d. The node contains three entries: `masks` 4 and 9 belong to `DATA_CATEGORY_1` (bit-pattern 01), and `mask` 2 refers to a `NODE` (bit-pattern 11). The intermediate results are listed below:

$$
\begin{aligned}
\mathit{unfilteredBitmap} &= 00_{31} \ldots \mathbf{01}_{9}\ 00\ 00\ 00\ 00\ \mathbf{01}\ 00\ \mathbf{11}\ 00\ 00_{0} \\
\mathit{bitmap} &= 00_{31} \ldots \mathbf{01}_{9}\ 00\ 00\ 00\ 00\ \mathbf{01}\ 00\ 00\ 00\ 00_{0} \\
(\mathit{marker} - 1) &= 00_{31} \ldots 00_{9}\ \mathbf{11}\ \mathbf{11}\ \mathbf{11}\ \mathbf{11}\ \mathbf{11}\ \mathbf{11}\ \mathbf{11}\ \mathbf{11}_{0} \\
\mathit{bitmap}\ \&\ (\mathit{marker} - 1) &= 00_{31} \ldots 00_{9}\ 00\ 00\ 00\ 00\ \mathbf{01}\ 00\ 00\ 00\ 00_{0}
\end{aligned}
$$

First, the bitmap is filtered according to the `type` that designates the group into which we want to index. After applying the filter, both `DATA_CATEGORY_1` entries remain, but the `NODE` entry is removed from the filtered bitmap.

Second, we create a marker, i.e., a single bit that is set to 1, at the position where the `mask`'s 2-bit pattern is stored in the bitmap. For tuple $F \mapsto 6$, the mask value of the key equals 9. The constant `1L` is shifted to the final marker position with the `mask` value that is scaled to 2-bit patterns (`mask << 1`). The marker is then used to create a another bit-mask that has all bits left of the `marker`'s position set to 1 (`marker - 1`).

Finally, the new bit-mask allows us to count all occurrences of `type` to the left of the position of $F \mapsto 6$ according to the total ordering within a group (`bitmap & (marker - 1)`). In our example, invoking `Long.bitCount` returns value 1, the resulting relative index of the tuple $F \mapsto 6$ within the `DATA_CATEGORY_1`. Note that the bit-count expression (line 4) also occurs in HAMT's simple compression. However, AXIOM generalizes this operation to support permutations by applying bitmap pre-processing and filtering, and offset scaling.

3.5 Roadmap of Evaluation

In our evaluation we will show that type-heterogeneous AXIOM multi-map offers better performance than comparable composite multi-maps in Scala and Clojure (Section 4), while generalizing with little overhead over state-of-the-art maps (Section 5). A real-world use-case (Section 6) underlines the effectiveness of type-heterogeneous AXIOM multi-maps when used as a graph for solving static analysis problems.

The source code of the data structures and benchmarks used in this paper is available online.[6] In order to counter concerns about internal validity, we would like to mention that the data structures presented in this paper are well tested and used in production in the runtime of a mature programming language.[7]

4 Case Study: Persistent Multi-Maps

We start by evaluating the performance characteristics of a type-heterogeneous AXIOM multi-map against idiomatic multi-map implementations in Clojure and Scala. Both languages are used widely in industry and the open-source world, and feature sophisticated and well engineered immutable collections in their standard libraries. Neither language provides native immutable multi-maps out of the box though, but nevertheless both languages suggest idiomatic solutions to transform normal polymorphic maps with nested sets into multi-maps.

VanderHart and Neufeld [27, p. 100–103] propose a solution for Clojure based on *protocols*, which are Clojure's solution for ad-hoc polymorphism that is similar to type classes in Haskell [18, 28]. Comparable to AXIOM, values are

[6] https://michael.steindorfer.name/papers/pldi18-artifact
[7] https://www.rascal-mpl.org

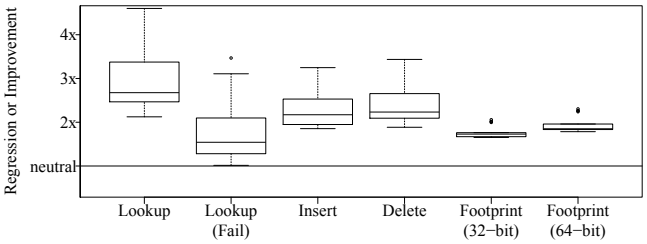


Figure 4. Performance comparison of an AXIOM multi-map against an idiomatic multi-map in Clojure (baseline).

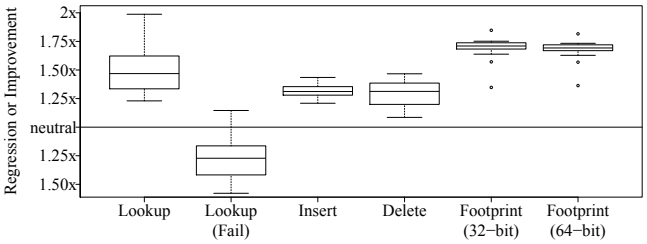


Figure 5. Performance comparison of an AXIOM multi-map against an idiomatic multi-map in Scala (baseline).

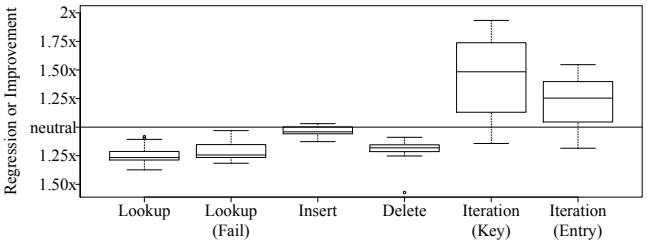


Figure 6. Performance comparison of an AXIOM multi-map against a CHAMP map (baseline).

stored either as a singleton or a nested set, however untyped. The multi-map protocol performs the necessary dynamic type checks and handles all possible case distinctions for lookup, insertion, and deletion.

Scala programmers would idiomatically use a trait for hoisting a regular map to a multi-map, by nesting typed sets as values of polymorphic maps. Scala's standard library unfortunately only contains a trait for mutable maps. We therefore ported the standard library's program logic of their mutable multi-map trait to the immutable case.

Assumptions. With the microbenchmarks of Sections 4 and 5 we evaluate the performance of individual operations in a controlled and synthetic setting that does not account for cost functions for `hashCode` and `equals` methods. The case study of Section 6 in contrast uses objects with costly `hashCode` and `equals` implementations.

4.1 Operations under Test

Insert: We call insertion in three bursts, each time with 8 random parameters to exercise different trie paths.[8] Firstly we provoke full matches (key and value present), secondly partial matches (only key present), and thirdly no matches (neither key nor value present). Next to insertion of a new key, this mixed workload also triggers promotions from singletons to full collections.

Delete: We call deletion in two bursts, each time with 8 random parameters. Provoking again full matches and partial matches. Next to deletion of a key, this mixed workload also triggers demotions from full collections to singletons, and compaction of trie nodes where applicable.

Lookup: We call lookup in two bursts to provoke full and partial matches. This test isolates how well the discrimination between singletons and full collections works.

Lookup (Fail): In a single burst with 8 random parameters we test unsuccessful lookups. We assume this test equivalent to Delete with no match.

To exercise if we can indeed exploit skewed distributions, we use for each size data point 50 % of 1 : 1 mappings and 50 % of 1 : 2 mappings. Although fixing the size of nested value sets may seem artificial, it allows us to precisely observe the singleton case, promotions to value sets, and demotions to singletons. The effect of larger value sets on memory usage and time can be inferred from that without the need for additional experiments.

4.2 Hypotheses

Hypothesis 1: We expect the performance of lookup, deletion, and insertion of an AXIOM multi-map to be equal or better to the competitors performance. AXIOM natively supports multi-maps in a space-efficient way and should execute faster than a hoisted multi-map in Clojure or Scala.

Hypothesis 2: Only for unsuccessful lookups we expect that AXIOM performs worse than Scala, based on results from related work [24] that explain the inherent differences between Scala's implementation and other hash-tries when it comes to memoizing hash codes.

Hypothesis 3: We expect average memory savings of at least 50 % compared to Scala, due to the omission of nested collections for singletons. We cannot assume space saving over Clojure in this regard since it also inlines singletons. Still we expect observable memory savings due to Clojure's simple compression that may contain empty array cells (in contrast to AXIOM's compression by permutation).

4.3 Experiment Setup

The benchmarks were executed on a computer with 16 GB RAM and an Intel Core i7-2600 CPU that has a base frequency of 3.40 GHz, 8 MB Last-Level Cache and 64 B cache lines. The software stack consisted of Fedora 20 operating

[8] For < 8 elements, we duplicated the elements until we reached 8 samples.

system (Linux kernel 3.17) and an OpenJDK (JDK 8u65) JVM. We disabled CPU frequency scaling and fixed the JVM heap sizes to 8 GB for benchmark execution.

To obtain statistically rigorous performance numbers, we adhere to best practices for (micro-)benchmarking on the JVM as for example discussed in Georges et al. [8], Kalibera and Jones [15]. We measure the execution time of the operations under test with the Java Microbenchmarking Harness (JMH), which is a framework to overcome the pitfalls of microbenchmarking. We configured JMH to invoke the Garbage Collector (GC) between measurement iterations to reduce a possible confounding effect of the GC on time measurements. For all experiments JMH performs 20 measurement iterations of one second each, after a warmup period of 10 equally long iterations. The precise numbers of benchmark iteration were obtained from tuning our benchmark setup to finish in reasonable time while still yielding accurate measurements with errors smaller than 1 %. Additional to the median runtime we report the measurement error as Median Absolute Deviation (MAD), which is a robust statistical measure of variability that is resilient to small numbers of outliers. Memory footprints of the data structure heap graphs are obtained at runtime with Google's *memory-measurer* library.[9]

Random Test Data Generation. In our evaluation we use collections of sizes 2^x for $x \in [1, 23]$, a size range which was previously used to measure the performance of hash-tries [2, 24]. For every size data point, we create a fresh collection instance populated with numbers from a random number generator. Random integers are used to model the distribution of the hash codes of the keys. In our case a uniform distribution from random integers models a good hash code implementation.

Protecting against Accidental Trie Shapes and Bad Memory Locations. We repeat every experiment for each size data point five times. Each time we use a different input tree that is generated from a unique seed. This counters possible biases introduced by the accidental shape of the tries, and accidental bad locations in main memory.

4.4 Experiment Results

Figures 4 and 5 show the relative differences of an AXIOM multi-map compared to the implementations in Clojure and Scala. Note that both figures use different x-axis scales due to diverging improvement factors. We summarize the data points of the runs with the five different trees with their medians. Each box plot visualizes the measurements for the whole range of input size parameters. We report improvements as speedups factors ($\text{measurement}_{\text{other}}/\text{measurement}_{\text{AXIOM}}$) above the neutral line, and degradations as slowdown factors below the neutral line, i.e., the inverse of the speedup equation. We now evaluate our hypotheses:

Confirmation of Hypothesis 1: Performance strictly improved over the competition, as expected. Lookup, Insert, and Delete perform by a median 1.47 x, 1.31 x, and 1.31 x faster than Scala, respectively. AXIOM clearly outperforms Clojure with a median speedups of 2.68 x, 2.17 x, and 2.23 x respectively for the aforementioned operations.

Confirmation of Hypothesis 2: As expected from related work studies, AXIOM performs worse than Scala for negative lookups. Runtimes increased by a median 1.27 x and at maximum by 1.58 x. Compared to Clojure, AXIOM improves performance of negative lookups by a median 1.54 x. Note that Clojure, like AXIOM, only stores hash-code prefixes and does not memoize the full hashes.

Confirmation of Hypothesis 3: Also, as expected, memory footprints improve by a median factor of 1.71 x (32-bit) and 1.69 x (64-bit) over Scala, and by a median factor of 1.73 x (32-bit) and 1.85 x (64-bit) over Clojure.

Even Smaller Footprints. We additionally compared two variants of AXIOM: one with fusion applied, and one that applied memory-layout specialization on top of fusion. In relation to both Clojure and Scala, memory footprints lowered on average by 2.43 x in the former setting, and by 5.1 x in the latter. Fusion had a strictly positive impact on execution times (due to less memory indirections), whereas specialization added a performance penalty of circa 20 %.

Discussion. We were surprised that the memory footprints of Clojure's and Scala's multi-map implementations are almost equal. From related work [24] we knew the typical trade-offs of both libraries: Scala mainly optimizes for runtime performance, while Clojure optimizes for low memory footprints. Code inspection revealed the cause of the relative improvement: Scala's hash-set does specialize singletons.

In this laboratory setup, an AXIOM multi-map resulted in improved runtimes of lookup, insertion, and deletion —with the notable exception of negative lookups when compared to Scala— while also significantly lowering memory footprints.

5 Case Study: Persistent Maps

In this section we evaluate the performance characteristics of AXIOM against CHAMP, a comparable data structure for immutable collections [24], to isolate the effects that are incurred by generalizing AXIOM towards type-heterogeneity that uses bitmap pre-processing and content histograms.

5.1 Operations under Test

To make AXIOM comparable to CHAMP, we use for each size data point 100 % of 1 : 1 mappings. The basic benchmarking setup and methodology is extensively outlined in Section 4. We test the same operations as in the previous section, however add Iteration (Key) and Iteration (Entry), which test the overhead of iterating through the map's keys and iteration through a flattened sequence of entries.

[9] http://openjdk.java.net/projects/code-tools/jmh/

5.2 Hypotheses

Hypothesis 4: We expect AXIOM's runtime performance of lookup, deletion, and insertion to be similar comparable to CHAMP's runtime performance, but never better. As AXIOM generalizes over specialized approaches such as CHAMP, running times should not degrade below a certain threshold; we feel that 25 % for median values and 50 % for maximum values would about be acceptable as a trade-off.

Hypothesis 5: Iteration over an AXIOM data structure is more complex than iterating over a regular map. However, due to the histogram abstraction, which early terminates when a node's payload is exhausted, we assume that the performance of AXIOM should be on a par with CHAMP.

Hypothesis 6: Memory footprints of AXIOM should in theory match CHAMP's footprints; both designs starkly differ in implementation choices, but they share the same 64-bit bitmap encoding overhead per node.

5.3 Experiment Results

Figure 6 reports for each benchmark the ranges of runtime improvements or degradations; we do conclude:

(Partial) Confirmation of Hypothesis 4: At all the basic operations, AXIOM's performance was strictly worse than the performance of the special-purpose CHAMP data structure. Lookup, Lookup (Fail), Insert and Delete degraded by a median 27 %, 24 %, 4 %, and 18 % respectively. All results stayed within the expected bounds, with exception of Lookup, which missed the target by 2 %.

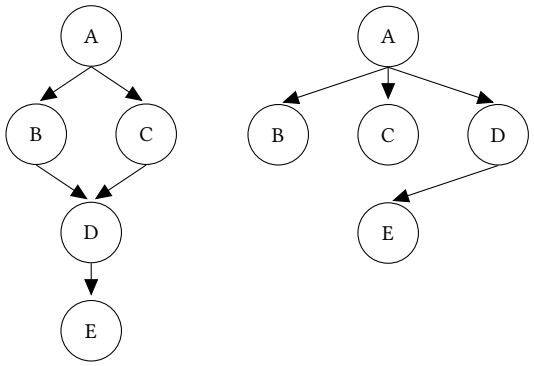
(Partial) Confirmation of Hypothesis 5: Counter to our intuition, both data structures yield different characteristics. AXIOM improved Iteration (Key) by a median 48 % and Iteration (Entry) by a median 25 %.

Confirmation of Hypothesis 6: As anticipated, AXIOM exactly matches the footprint of CHAMP. The footprint data points were therefore elided from Figure 6.

Discussion. When used as a map, AXIOM achieves acceptable runtimes across all tested operations, although it clearly lags behind the special-purpose CHAMP map. Especially for lookup and deletion, processing of 2-bit states and bitmap filtering does add up, whereas insertion seems only slightly affected. On the positive note, the histogram abstractions for batch-processing of AXIOM trie nodes positively influence performance of key- and key/value pair iteration even for the map use case (average speedups between 25–48 %).

6 Case Study: Static Program Analysis

The microbenchmarks of the previous two sections help to separate individual factors that influence AXIOM's performance, but they do not show the support for the expected improvements on an algorithm "in the wild". To add this perspective, we selected computing control-flow dominators using fixed point computation over collections of nodes [1].



(a) Control-Flow Graph **(b)** Dominator Tree

Figure 7. Example of a control-flow graph (a) and its dominator tree (b) that is derived from the dominance equations.

Dominators are widely used in practice [9], e.g., in compilers for structural analysis and detection of natural loops, and to calculate control dependencies and program dependence graphs [12] for purposes of program analysis or optimization.

Although we do not claim the algorithm in this section to be representative of all applications of multi-maps, it is a basic implementation of a well known and fundamental algorithm in program analysis. It has been used before to evaluate the efficiency of hash-trie set and map implementations [24], where sets were nested as the values a polymorphic map for simulating multi-maps with basic collection types.

Shape of Data and Data Set Selection. The nodes in the control-flow graph graphs are complex recursive ASTs with arbitrarily expensive (but linear) complexity for `hashCode` and `equals`. More importantly, the effect of type-heterogeneous AXIOM's multi-map encoding does depend on the accidental shape of the data, as it is initially produced from the raw control-flow graphs, and as it is dynamically generated by the incremental progression of the algorithm. Figure 7 shows a simple example of a control-flow graph (a) and its dominator tree (b) that is derived from the following equations:

$$\mathrm{Dom}(n_0) = \{n_0\}$$

$$\mathrm{Dom}(n) = \left(\bigcap_{p \in \mathrm{preds(n)}} \mathrm{Dom}(p) \right) \cup \{n\}$$

As visible from the example, both the control-flow graph and the dominator tree representation freely mix $1:1$ and $1:n$ mappings and exercise AXIOM's type-heterogeneity as well. We implemented the two dominance equations directly on top of the multi-maps. The Dom and preds relations are implemented as multi-maps, instead of using sets as the values of polymorphic maps. Our code uses projections, and set union and intersection in a fixed-point loop. The big intersection is not implemented directly, but staged by first

Table 1. Runtimes of AXIOM for the CFG dominators experiment per CFG count, and statistics over preds relation about shape of data (unique keys, tuples, 1 : 1 mappings).

#CFG	CHAMP	AXIOM	#Keys	#Tuples	% 1 : 1
4096	173 s	174 s	315 009	331 218	91 %
2048	84 s	85 s	162 418	170 635	91 %
1024	64 s	62 s	88 952	93 232	92 %
512	28 s	28 s	43 666	45 743	92 %
256	19 s	18 s	21 946	22 997	92 %
128	14 s	14 s	13 025	13 583	93 %

producing a set of sets for the predecessors and intersecting the respective sets with each other.

For our experiment, we used a data set that contains the complete control-flow graphs for all units of code of the widely-used content management system *Wordpress*.[10] The data set file has the size of 260.6 MB in serialized form and origins from a corpus that is used to analyze PHP applications [13]. The data set contains ±5000 control-flow graphs for all units of code (function, method and script). We consider the size of the data sufficient for our evaluation, because the algorithm of computing dominators equations is mostly CPU-bound. Relative space savings due to AXIOM's encoding should translate to bigger data sets as well. One main goal of this evaluation is show the applicability of AXIOM to shapes of data that occur in real life.

Like before, we used JMH to measure execution time. We ran the dominator calculations on a random selection of the aforementioned graphs. The set of selected graphs range between a size of from 128 to 4096 in exponential steps. We measured as well the number of many-to-one and many-to-many entries in the multi-maps.

Hypotheses. On the one hand, since the Dom relation is expected to be many-to-many with large value sets it should not generate any space savings but at least it should not degrade the runtime performance either compared to CHAMP (Hypothesis 7). On the other hand we expect the preds relation to be mostly one-to-one and we should get good benefit from the inlining of singletons (Hypothesis 8).

Results. Table 1 shows that the mean runtimes for CHAMP and AXIOM are almost identical (at most ±2 s difference). Due to equal runtimes, AXIOM retains the same magnitude of speedups that CHAMP yielded over Clojure and Scala [24], from minimal 9.9 x to 28.1 x. We also observed that the shape of data in the preds relation contains a high number of 1 : 1 mappings (median 92 %) and that the average ratio of unique keys to tuples is 1.05 x. The outcome suggests that a type-heterogeneous AXIOM multi-map is an excellent choice for reverse indices. In the case of control-flow graphs edges frequently flow into a set of nodes, however in an inversed graph the parent of a node is mostly unique (cf. Figure 7).

We conclude both Hypothesis 7 and Hypothesis 8 to be confirmed. In the case of Wordpress, the CFG algorithm turns out to profit over CHAMP in terms of memory savings from AXIOM's optimizations for 1 : 1 mappings. The preds relation compresses by 4.4 ×, from 37.7 MB (CHAMP) to 8.4 MB (AXIOM) even without specialization applied. In this case, the algorithmic experiment is dominated by efficiency of set and relational algebra operations and memory turnover in the fixed-point loop, rather than the memory footprint of the preds relation (which is calculated outside of the loop).

7 Related Work

Efficient Immutable Hashtables based on Prefix Trees. The two contenders for implementing efficient unordered hashtables for collection libraries, HAMT [2] and CHAMP [24], were discussed in Section 2. Although not reaching peak performance of CHAMP, AXIOM generalizes over CHAMP with low overhead and introduces with histograms an abstraction that scales to multiple content categories. Therefore, AXIOM is flexible enough to implement a range of efficient type-heterogeneous data structures.

Reducing the Memory Footprint of Collections is a goal of other people as well. Steindorfer and Vinju [2014] specialized internal trie nodes to gain memory savings of 55 % for maps and 78 % for sets at median while adding 20–40 % runtime overhead for lookup. Their approach minimized the amount of specializations to mitigate effects on code bloat and run-time performance. Gil and Shimron [2012] identified sources of memory inefficiencies in Java's mutable collections and proposed compaction techniques to counter them. They improved memory footprints of Java's `Hash{Map,Set}` and `Tree{Map,Set}` data structures by 20–77 %. We observed that through added heterogeneity, AXIOM by design allows to dispatch on various internal memory representation of payload to significantly reduce memory footprints.

Trie Data Structures for In-Memory Databases. Leis et al. [16] proposed the Adaptive Radix Tree (ART), a general purpose index structure for in-memory databases that —instead of hashing— encodes the data itself in the prefix tree, similar to AMTs. Hashing is avoided by ART because it only allows point queries (due to unordered storage) and does not support range queries. ART represents an ephemeral and mutable data storage, AXIOM in contrast is tailored towards immutable data representations. ART uses 8-bit prefix masks for implementing an 256-ary tree, compared to AXIOM's 5-bit prefix masks that yield a 32-ary tree. For immutable collections, 5-bit prefixes experimentally yield good performance balance

[10] https://wordpress.com

between search and update operations [3]. ART internally offers four different node sizes (4, 16, 48, 256) to less frequently resize nodes. In the context of collections, such an approach wastes memory due to common empty cells. In contrast, AXIOM resizes on each immutable update (excluding the transient case) and does not contain empty cells at all. While basic concepts of both trie-based approaches overlap, the chosen trade-offs clearly address different use cases. ART aims to be a (mutable) general purpose index structure, however forgoes hashtable indexing performance and compact representation. AXIOM in contrast allows the implementation of mutable and immutable unordered hashtables that are memory efficient, and more specifically aims to be a flexible template for type-heterogeneous collection data structures.

Variability of Features in the Domain of Collections. Steindorfer and Vinju [25] performed a preliminary study about feature variability in collection data structures and implementation choices for implementing those features. The authors presented a concise framework for implementing various purely functional collection data structures, and further covered a software product line that relieves collection library designers from optimizing for the general case. While this paper was partly inspired by the findings of Steindorfer and Vinju, we contribute a novel and sophisticated type-heterogeneous data structure that would subsume multiple points in their presented solution space. AXIOM is suitable for handwritten collection libraries that supply a minimal set of efficient abstractions while maximizing utility.

8 Conclusion

We proposed AXIOM, a new design for hash-array mapped tries that strictly generalizes previous solutions by Bagwell [2] and Steindorfer and Vinju [24]. AXIOM allows the implementation of efficient map and multi-map data structures. When used as a map, AXIOM is on average between 4–27 % slower at lookup, insertion, and deletion than the most efficient immutable hashtables we tested, while offering better iteration performance (25–48 %). AXIOM's performance shines when compared to idiomatic multi-maps in Clojure and Scala: runtimes are usually on a par or better, while significantly saving memory (1.87–5.1 x, depending on the configuration). We note that a type-heterogeneous AXIOM multi-map is especially suited for reverse indices (e.g., of control-flow graphs) that turn many-to-many content into mostly many-to-one content, while efficiently catering for a few many-to-many cases.

Acknowledgments

This research has been supported by the NWO TOP-GO grant #612.001.011 *"Domain-Specific Languages: A Big Future for Small Programs"* and the NWO VICI project #639.023.206 *"Language Designer's Workbench"*.

References

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. 1986. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley.
[2] Phil Bagwell. 2001. *Ideal Hash Trees.* Technical Report LAMP-REPORT-2001-001. Ecole polytechnique fédérale de Lausanne.
[3] Phil Bagwell and Tiark Rompf. 2011. *RRB-Trees: Efficient Immutable Vectors.* Technical Report EPFL-REPORT-169879. Ecole polytechnique fédérale de Lausanne.
[4] Carl Friedrich Bolz, Lukas Diekmann, and Laurence Tratt. 2013. Storage Strategies for Collections in Dynamically Typed Languages. In *OOPSLA '13.* ACM.
[5] Rene de la Briandais. 1959. File Searching Using Variable Length Keys. In *IRE-AIEE-ACM '59 (Western).* ACM.
[6] James R. Driscoll, Neil Sarnak, Daniel D. Sleator, and Robert E. Tarjan. 1986. Making Data Structures Persistent. In *STOC '86.* ACM.
[7] Edward Fredkin. 1960. Trie Memory. *Commun. ACM* 3, 9 (1960).
[8] Andy Georges, Dries Buytaert, and Lieven Eeckhout. 2007. Statistically Rigorous Java Performance Evaluation. In *OOPSLA '07.* ACM.
[9] Loukas Georgiadis, Robert E. Tarjan, and Renato F. Werneck. 2006. Finding Dominators in Practice. *J. Graph Algorithms Appl.* 10, 1 (2006).
[10] Joseph Gil and Yuval Shimron. 2012. Smaller Footprint for Java Collections. In *ECOOP '12.* Springer-Verlag.
[11] Eiichi Goto. 1974. *Monocopy and Associative Algorithms in Extended Lisp. University of Toyko.* Technical Report.
[12] Mark Harman, David Binkley, Keith Gallagher, Nicolas Gold, and Jens Krinke. 2009. Dependence Clusters in Source Code. *ACM Trans. Program. Lang. Syst.* 32, 1 (2009).
[13] Mark Hills and Paul Klint. 2014. PHP AiR: Analyzing PHP systems with Rascal. In *CSMR/WCRE '14 Tools.* IEEE.
[14] Atsushi Igarashi and Mirko Viroli. 2002. On Variance-Based Subtyping for Parametric Types. In *ECOOP '02.* Springer-Verlag.
[15] Tomas Kalibera and Richard Jones. 2013. Rigorous Benchmarking in Reasonable Time. In *ISMM '13.* ACM.
[16] V. Leis, A. Kemper, and T. Neumann. 2013. The adaptive radix tree: ARTful indexing for main-memory databases. In *ICDE '13.*
[17] Chris Okasaki. 1999. *Purely Functional Data Structures.* Cambridge University Press.
[18] Simon Peyton Jones. 2003. The Haskell 98 Language and Libraries. *Journal of Functional Programming* 13, 1 (2003).
[19] Juan Pedro Bolívar Puente. 2017. Persistence for the Masses: RRB-vectors in a Systems Language. *Proc. ACM Program. Lang.* 1, ICFP, Article 16 (2017).
[20] Neil Sarnak and Robert E. Tarjan. 1986. Planar Point Location Using Persistent Search Trees. *Communications of the ACM* 29, 7 (1986).
[21] Siddhartha Sen, Robert E. Tarjan, and David Hong Kyun Kim. 2016. Deletion Without Rebalancing in Binary Search Trees. *ACM Transactions on Algorithms* 12, 4, Article 57 (2016).
[22] Lukas Stadler. 2014. *Partial Escape Analysis and Scalar Replacement for Java.* Ph.D. Dissertation. Johannes Kepler University Linz.
[23] Michael J. Steindorfer and Jurgen J. Vinju. 2014. Code Specialization for Memory Efficient Hash Tries (Short Paper). In *GPCE '14.* ACM.
[24] Michael J. Steindorfer and Jurgen J. Vinju. 2015. Optimizing Hash-array Mapped Tries for Fast and Lean Immutable JVM Collections. In *OOPSLA '15.* ACM.
[25] Michael J. Steindorfer and Jurgen J. Vinju. 2016. Towards a Software Product Line of Trie-based Collections. In *GPCE '16.* ACM.
[26] Nicolas Stucki, Tiark Rompf, Vlad Ureche, and Phil Bagwell. 2015. RRB Vector: A Practical General Purpose Immutable Sequence. In *ICFP '15.* ACM.
[27] L. VanderHart and R. Neufeld. 2014. *Clojure Cookbook: Recipes for Functional Programming.* O'Reilly Media.
[28] P. Wadler and S. Blott. 1989. How to Make Ad-hoc Polymorphism Less Ad Hoc. In *POPL '89.* ACM.