Efficient Immutable Collections

Efficient Immutable Collections

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Universiteit van Amsterdam
op gezag van de Rector Magnificus
prof. dr. D.C. van den Boom
ten overstaan van een door het college voor promoties
ingestelde commissie,
in het openbaar te verdedigen in de Agnietenkapel
op {weekdag} {dag} {maand} 2017, te {uur}.00 uur

door

Michael J. Steindorfer

geboren te Friesach

Promotiecommissie

1e Promotor:	Prof. dr. P. Klint
2e Promotor:	Prof. dr. J.J. Vinju
Overige leden:	Prof. dr. P. van Emde Boas Prof. dr. M.L. Kersten Prof. dr. M. Püschel Dr. C.U. Grelck Dr. D. Syme

Faculteit der Natuurwetenschappen, Wiskunde en Informatica

The work in this thesis has been carried out at Centrum Wiskunde & Informatica (CWI) under the auspices of the research school IPA (Institute for Programming research and Algorithmics)

Contents

Chapter 1

Introduction

> *"Good programmers worry about data structures and their relationships."*
>
> — Linus Torvalds

> *"Good programmers worry about their relationships and data structures."*
>
> — the author

Using an object-oriented or functional programming language without data structures is comparable to riding an electric car without batteries included. The car would still operate with a power cord attached, but user experience would be poor and utility would be limited. Programming languages gain utility from processing and transforming data being hold by *data structures*. We use the following definition of the term data structure [PB04]:

> *"An organization of information, usually in memory, for better algorithm efficiency, such as queue, stack, linked list, heap, dictionary, and tree, or conceptual unity, such as the name and address of a person. It may include redundant information, such as length of the list or number of nodes in a subtree."*
>
> *"Most data structures have associated algorithms to perform operations, such as search, insert, or balance, that maintain the properties of the data structure."*

Data structures are a key ingredient to enable the design of efficient algorithms. To provide programmers with a versatile toolbox, most programming languages do include commonly used data structures in their standard libraries. This attitude is also known as the *"batteries included"* philosophy and refers to *"having a rich and*

versatile standard library which is immediately available, without making the user download separate packages".* An essential part of standard libraries are *collections*. A collection is a data type that *"represents a group of objects, known as its elements"*.† Common collection data types —such as list, set, or map— under-pin many, if not most, applications in general purpose programming languages and therefore constitute an essential part of standard libraries.

Generally speaking, collection data structures are general purpose implementations that should carefully balance operation runtimes and memory footprints. Those implementations are not optimized for a certain problem size, they are assumed to perform well when containing a few bytes of data, or gigabytes of data. In order to balance the properties of general purpose data structures, engineers who design standard library data structure have to solve a multi-objective optimization challenge by making trade-offs. Unfortunately the objectives of *generality* and *performance* rival with each other, resulting in data structures with mediocre performance for specific contexts or usage patterns. E.g., several collection data structures can be implemented more efficiently without supporting a deletion operation. Because the predefined set of supported operations on collections is fixed, client code that does not depend on deletion essentially uses a suboptimal implementation.

In this thesis we introduce data structures that are more performant than comparable state-of-the-art standard library data structures —in terms of operation runtimes and memory footprints— and more general by supporting storage and retrieval of type-heterogeneous data‡ where parametric polymorphism [OW97] on the language level falls short. We specifically target *immutable*, *unordered* and *hashed* data types, such as widely-used hash-sets and hash-maps; we will specify the concepts just mentioned as well as the characteristics of the proposed data structures throughout the remainder of this chapter.

Industrial and Practical Impact. We start with the conjecture that even widely used data structures still contain a large potential for optimization, which proved to be true. By targeting widely used data structures, the contributions of this thesis could have a high practical and industrial impact.

This thesis describes data structure implementations, written in Java, that run on and are optimized for the Java Virtual Machine (JVM).§ We chose the JVM as platform, because there exist industry grade JVM implementations, such as Oracle's HotSpot™,¶ that are open-sourced, maintained by strong industrial institutions, and have been optimized for more than a decade. Furthermore, the JVM hosts a multitude

*`https://www.python.org/dev/peps/pep-0206/`

†`http://docs.oracle.com/javase/8/docs/api/java/util/Collection.html`

‡In type-heterogeneous data structures, elements could have different (internal) representations.

§cf. *The Java Virtual Machine Specification*: `http://docs.oracle.com/javase/specs/jvms/se8/html/`

¶`http://openjdk.java.net/groups/hotspot/`

```
interface List<E> {
  int size();
  void add(E item);
  E get(int index);
}
```

Listing 1.1: Object-oriented interface of a `List` data type.

of programming language runtimes,[‖] including many that feature interoperability with Java. By choosing Java, our research results are immediately available to many programming language runtimes and standard libraries.

1.1 Collection Libraries

Most modern programming languages support the creation of modules or libraries to encapsulate reusable data types that can be shared amongst different programs. The Java programming language ships with a standard library that features a rich set of *collection* data types.

The technical term *collection* describes standard library data structures of commonly used data types for representing groups of objects [NW06]. The semantics of the various forms that groups of objects can take (cf. Section "Variability Dimensions of Collections") are modeled by *abstract data types* and are exposed as a hierarchy of interfaces. Each interface is typically matched by one or more concrete *data structure* implementations that vary in performance characteristics. Abstract data types express a contract of *what* functionality a data type must support, without dictating a data structure implementation *how* to accomplish such a contract. Listing 1.1 illustrates the interface of a simple `List` collection data type that supports querying the amount of elements it holds (`size`), appending elements at the end (`add`), and retrieving elements based on their position (`get`).

Data structures, implementing object-oriented interfaces, encapsulate their internal state and hide implementation details of the methods operating on the state. E.g., when implementing the `List` interface of Listing 1.1, an `ArrayList` (cf. Listing 1.2) may use Java's built-in array primitive to compactly store the collection's elements while a `LinkedList` (cf. Listing 1.3) may use individual interlinked `Node` objects, each containing references to their next and previous element. How an interface is implemented, has direct impact on the expected performance of an operation. Lookup (`get`) executes in constant time on an `ArrayList` regardless of the size of the data structure due to random-access support of the underlying array, while lookup executes in (worst case) linear time on a `LinkedList`.

[‖] `http://jvmlangsummit.com`

```
abstract class ArrayList<E> implements List<E> {
  // internal representation
  Object[] elementData;
}
```

Listing 1.2: Internal state representation of an `ArrayList` data type.

```
abstract class LinkedList<E> implements List<E> {
  // internal representation
  int size;
  Node<E> first;
  Node<E> last;
}

abstract class Node<E> {
  // internal representation
  E item;
  Node<E> next;
  Node<E> prev;
}
```

Listing 1.3: Internal state representation of a `LinkedList` data type.

Substitutability. In object-oriented programming languages, programmers are allowed to program against the interface of a data type —an *opaque* abstraction— or against a specific implementation —a *transparent* abstraction. The *Liskov substitution principle* [LW94; LW99] states that if two types S and T are in a sub-type relationship —S is a subtype of T— then all objects of the super-type T may be replaced by objects of the more specific sub-type S, without altering a program's correctness. Substitutability describes a form of strong behavioral sub-typing that is usually supported by collection libraries.

Java's collection library exposes collection data types as Java interfaces and furthermore contains at least one or more data structure implementations per interface. Substitutability allows replacing usages of the interfaces by usages of specific implementations in the source code. To regulate behavioral sub-typing, Java's collection library contains a range of documented design choices that potentially impact the performance of data structure implementations. E.g., the definition of `hashCode` of the `java.util.Map` interface assumes a row-wise data decomposition of the tuples, and complicates incremental computations on column-wise decompositions.

Parametric Polymorphism. Next to distinguishing between interfaces and implementations, collections in Java and C# abstract over their element types with a language feature named *generics*, a form of parametric polymorphism [OW97; KS01].

Generics were introduced in Java SE 5 to allow the parametrization of classes by type variables that act as placeholders. Generics allow devising code that can be re-used and checked at compile time for type-safety. Listing 1.4 illustrates a `Box` data type that supports storage and retrieval of a single untyped object. While the `Box` supports arbitrary reference types, it is not type safe and requires unchecked type casts, which could lead to latent program errors. Without generic type arguments, a Java programmer interested in type-safety would be inclined to specialize the data type for a specific reference type or primitive type (cf. Listing 1.5). An equivalent data type using generic type arguments —offering code re-use without duplication and compile-time type safety— is shown in Listing 1.6.

Unfortunately the current version of Java does not support generics over primitive types, which complicates the design of generic and performant collections that contain primitive data or that mix primitive data with object instances.

1.2 Variability Dimensions of Collections

Collection libraries are often organized by the intended usage of the contained collections. E.g., collection libraries are split and specialized by several of the following criteria [OM09] in the JVM languages Java and Scala:

Split by data type semantics: Interfaces and implementations for a range of abstract data types such as lists, sets, bags, maps, multi-maps, and so forth.

Split by ordering: The elements of a collection are either ordered arbitrarily or non-arbitrarily. The semantics of some collections (e.g., set) do not dictate a particular ordering. Hashing may arbitrarily shuffle the element order. In contrast, the semantics of some collections dictate non-arbitrary ordering, e.g., alphabetic ordering of strings in a sorted-set, or sorting by temporal properties such as insertion order in a linked hash-set.

Split by update semantics: Data structures either allow mutation of their content over time, or they are immutable. So called *transient* data structures represent the middle ground by allowing efficient batch updates on otherwise immutable data structures.

Split by processing semantics: Data structures either support sequential, parallel (e.g., by splitting and merging data), or concurrent processing.

```
class Box {
  private final Object item;

  public Box(Object item) { this.item = item; }
  public Object get()     { return item;      }
}

// EXAMPLE:
//
// Box boxedItem = new Box("example");
// String item = (String) boxedItem.get(); /* requires unchecked cast to String */
```

Listing 1.4: A basic data type supporting storage and retrieval of *untyped* objects.

```
class IntBox {
  private final int item;

  public IntBox(int item) { this.item = item; }
  public int get()        { return item;      }
}

// EXAMPLE:
//
// IntBox boxedItem = new IntBox(13);
// int item = boxedItem.get(); /* no cast required, however restricted to int */
```

Listing 1.5: A specialized data type supporting storage and retrieval of **`int`** values.

```
class GenericBox<T> {
  private final T item;

  public GenericBox(T item) { this.item = item; }
  public T get()            { return item;      }
}

// EXAMPLE:
//
// GenericBox<String> boxedItem = new GenericBox<>("example");
// String item = boxedItem.get(); /* generically supports all reference types */
```

Listing 1.6: A generic data type supporting storage and retrieval of *typed* objects.

Split by implementation strategy: Different implementation strategies 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.

The dimensions span a multi-dimensional space with a large number of variants. In contrast to the large domain with a large variability, collection libraries of programming languages typically provide only a very small subset of the most frequently demanded data structures.

From an implementer's perspective, in terms of effort it is often not feasible to manually implement and maintain a complete set of variants. Because of the slightly different requirements and optimizations, each data structure variant usually comes with a separate hand-written implementation that evolves individually.

From a user's perspective, it is a double-edged sword to have many implementations per data type available. On the one hand, expert users could benefit from a wide range of specialized data structures (though selection requires knowledge about the options and their trade-offs). On the other hand, collection libraries accommodate novice users with exposing a small set of implementations. E.g., often users are just interested in a particular abstract data type without requiring particular performance guarantees or knowledge about implementation details.

In order to satisfy both the implementer's and the user's perspective, collection libraries traditionally settle for trade-offs between *generality* and *performance*: they aim to be as powerful as possible while keeping the code base simple and slim, at the cost of giving up maximum efficiency.

We expect that it is feasible to use generative programming in the traditional sense [McI68; Big98; CE00] to overcome current trade-offs and to satisfy both the implementer's and the user's perspective. Analyzing and understanding the domain, its variability, and engineering and design trade-offs are the key requirements for success. We further expect that by constraining the domain of collections, by fixing the update semantic to *immutable*, it is possible to find a small set of intermediate code generator abstractions that efficiently cover the chosen sub-domain.

Research Question: What are the commonalties and differences between different data structures in the domain of collection libraries? Can we generalize over the sub-domain of immutable collections by modeling variability and documenting implementation choices? *Research Question (RQ-1). Addressed by Chapter 2.*

1.3 Immutable Collections

Immutability is a characteristic that can be associated with arbitrary objects and —in the context of this thesis— with collections in particular. In its section about concurrency, Oracle's *The Java™ Tutorials** series describes what immutability is and what to expect from immutable objects, especially in concurrent environments:

> *"An object is considered immutable if its state cannot change after it is constructed. Maximum reliance on immutable objects is widely accepted as a sound strategy for creating simple, reliable code.*
>
> *Immutable objects are particularly useful in concurrent applications. Since they cannot change state, they cannot be corrupted by thread interference or observed in an inconsistent state.*
>
> *Programmers are often reluctant to employ immutable objects, because they worry about the cost of creating a new object as opposed to updating an object in place. The impact of object creation is often overestimated, and can be offset by some of the efficiencies associated with immutable objects. These include decreased overhead due to garbage collection, and the elimination of code needed to protect mutable objects from corruption."*

In general, immutable data structures [Oka99] are applied in domains such as functional languages [CD08; Vis04; KvdSV09], proof assistants [BJM13], meta-programming [KvdSV09], algebraic specification formalisms [vdBra+02; Bor+98; Cla+01; Mos04; Bal+07], and compiler infrastructures [Vaz+07]. Implementations of logic programming formalisms like, Datalog [HVdM06], can also benefit from optimizations of immutable data-structures [Ram+95].

Immutable collections are a specific area most relevant to functional or hybrid functional/object-oriented programming such as practiced by Clojure†, Rascal‡, and Scala§ programmers on the JVM.

In the book *Effective Java* [Blo08], Joshua Bloch said that *"immutable [collections] provide many advantages, and their only disadvantage is the potential for performance problems under certain circumstances"*. While state-of-the-art immutable collections do have advantages over mutable collections, the property of immutability comes at the cost of decreased performance of operations like lookup, insertion, and deletion. These performance differences arise because state-of-the-art immutable collections are implemented as shallow trees, while mutable collections are based on flat continuous arrays. Hence, immutable collections are challenging to optimize.

*https://docs.oracle.com/javase/tutorial/essential/concurrency/immutable.html
†http://clojure.org
‡http://rascal-mpl.org
§http://scala-lang.org

Goal Definition. In the bigger picture the overall goal of this thesis is to strengthen the performance perspective and expressivity of immutable data structures, such that they could become viable default choices over their mutable counterparts. This goal can be formulated as follows:

> *The goal of this research is to make immutable data structures as fast as mutable data structures, or even faster, while maintaining the beneficial property of immutability.*

Making immutable collections efficiently available in object-oriented programming languages transfers some of the benefits of functional and formal languages to mainstream programming: immutability is a foundation for equational reasoning, i.e., reasoning about your source code as it was a mathematical expression [HO80].

Immutability from a User's Perspective. Object-oriented data structures typically allow modifying (or mutating) their in-memory content. Instantiation of a mutable data type yields a stable reference to the data structure. The reference does not change, however, the internal state of objects may change by invoking methods on the object. Listing 1.7 exemplifies the usage of a mutable data type, as contained in Java's collection library. Multiple state modifications, to add new elements, are listed as separate imperative statements. This style of Application Program Interface (API) is common for mutable object-oriented libraries.

In contrast, immutable data structures prohibit internal modification. The content of a data structure is guaranteed to not change. However, immutable data structures yet allow *updates* by returning new instances upon modification. To enable updates, immutable data structure expose alternative APIs to the users. Listing 1.8 illustrates the usage of a purely functional API. The `add` operation is implemented as a function that takes two arguments, a set and an element to be added, and returns a reference to a newly constructed set containing the added element. A purely functional programming style, requires chaining of the calls to `add` as nesting expressions. Libraries of object-oriented languages do usually provide an object-oriented fluent interface on top of functional data structures, as shown in Listing 1.9. Hence, API design requires different considerations for mutable and immutable data structures in object-oriented languages.

Immutability from the Implementation's Perspective. From the user's perspective, immutability reflects the simple guarantee that the content of a data structure does not change. Deriving new *updated* instances from existing data structures does not give the user guarantees about the performance to be expected and the implementation of such a derivation operation. E.g., it is not obvious how much

```
Set<Integer> mutableSet = new HashSet<>();
mutableSet.add(1);
mutableSet.add(2);
mutableSet.add(3);
mutableSet.add(4);
```

Listing 1.7: API usage of a mutable `Set` data structure.

```
ImmutableSet<Integer> set = add(add(add(add(ImmutableSet.empty(), 1), 2), 3), 4);
```

Listing 1.8: Functional API usage of a immutable set data structure.

```
ImmutableSet<Integer> set = ImmutableSet.empty()
  .add(1)
  .add(2)
  .add(3)
  .add(4);
```

Listing 1.9: Fluent API usage of an immutable set data structure.

data has to be copied when executing deriving a new instance from a large data structure when adding a new element. A naive attempt is shown in Listing 1.10. First, a copy requiring linear space is allocated, second, it temporarily mutates the copy, before finally freezing the copy to avoid further mutation. Especially when dealing with potentially big data structures, requiring a linear space copy at each modification renders this naive effort inefficient for most if not all algorithms.

1.4 Persistent Data Structures

An advanced implementation technique for immutable data structures makes use of a property that is commonly known as *persistency* to achieve more efficient data structure derivations [Got74; Dri+86; Oka99]. Please note that the term persistency has nothing in common with the likewise named property in the context of database systems. We define the term *persistent data structure* as follows:

> *A persistent data structure is an immutable data structure that is a Directed Acyclic Graph (DAG) and consists of separate immutable segments. A persistent data structure is incrementally constructed by linking new immutable data segments to an existing DAG of immutable data structure segments.*

```
// allocating new instance
Set<Integer> __tmp = new HashSet<>(hugeSet.size + 1);

// temporary mutation
__tmp.addAll(hugeSet);
__tmp.add(34);

// freezing state
Set<Integer> updatedSet = Collections.unmodifiableSet(__tmp);
```

Listing 1.10: Immutable updates by naive cloning (`addAll`) requiring linear space.

In contrast to mutable data structures, instances of persistent data structure can safely share common (immutable) DAG sub-structures. Incrementally constructing a persistent data structure, is similar to versioning or snapshotting states of the data structure over time. Driscoll et al. [Dri+86] characterize persistency as follows:

> *"Ordinary data structures are ephemeral in the sense that a change to the structure destroys the old version, leaving only the new version available for use. In contrast, a persistent structure allows access to any version, [...] at any time."*

A persistent data structure therefore has many things in common with regular immutable data structures: they allow for new versions while leaving the previous versions untouched. However, the implementation strategies for persistent data structures differ. Persistent data structures mostly express derivations of data structures as small deltas (i.e., new immutable segments) to their previous states. Thus, only a small fraction of the data structure has to be copied, while structurally sharing and linking back to the parts of the data that remain unmodified. Nowadays, since Okasaki's book on *Purely Functional Data Structures* [Oka99], the term persistency is synonymously used for immutable data structures with structural sharing.

In practice, tree data structures in particular are suitable for expressing persistent delta updates. However, the most basic persistent data structure is the *cons-list*, a wide-spread data structure that was present in most LISP-like interpreters [McC60] early on. In the following, we will first introduce a basic persistent cons-list, before outlining the state-of-the-art of tree-based persistent data structures, relevant for immutable collections.

Cons-List. The term *cons-list* originates form the LISP programming language [McC60]. In most LISP dialects, `cons` is an essential function that constructs memory cells holding ordered pairs of values, or references to values. Those cons-cells are then used to construct more sophisticated data structures, such as single-linked lists, or trees. Cons-cells are the foundation for derived persistent data structures.

```
abstract class ConsList<E> {

  /* FIELD ACCESSORS */
  public abstract E head();
  public abstract ConsList<E> tail();

  /* DATA TYPE PROPERTIES */
  public abstract boolean isEmpty();
  public abstract int size();

  /* LIST CONSTRUCTION */
  public static final <E> ConsList<E> empty() { return EMPTY_LIST; }
  public ConsCell<E> cons(E item) { return new ConsCell<>(item, this); }

  /* DATA TYPE IMPLEMENTATIONS */
  private static final ConsList EMPTY_LIST = new ConsList() { /* omitted */ };
  private static final class ConsCell<E> extends ConsList<E> { /* omitted */ }
}
```

Listing 1.11: Internal state representation of an immutable `ConsList` data type.

Listing 1.11 shows an object-oriented implementation of an immutable `ConsList` data type in Java. The `ConsList` data type represents a sequence of ordered tuples of the properties `head` and `tail`. The `ConsList` is an Algebraic Data Type (ADT), that is a composite of two implementations: Lists are incrementally constructed by prepending new `ConsCell` objects (cf. Listing 1.13) to the `EMPTY_LIST` constant (cf. Listing 1.12), which is the terminator of list instances.

Listing 1.13 depicts the source code of the `ConsCell` implementation, that stores the local properties `head` and `tail` in fields. Both field references are immutable by using the keyword **`final`**. Most notably, the implementations of the methods `size` and `hashCode` are recursive over the list, and end when encountering the `EMPTY_LIST`.

To recapitulate, a `ConsList` is a persistent data structure that consists of `ConsCell` segments. New lists are constructed by prepending new `ConsCell` segments to an existing list, without destroying the old list. Every segment within a list designates a valid (shorter) previous version of the list. Such immutable single-linked lists are still present nowadays in standard libraries of programming languages. E.g., on the JVM, Clojure contains an immutable sequence data type, named `seq`,* and Scala contains an immutable `List`† data type.

*`https://clojure.org/reference/sequences`

†`https://www.scala-lang.org/api/current/#scala.collection.immutable.List`

```
1  private static final ConsList EMPTY_LIST = new ConsList() {
2
3    public Object head() { throw new UnsupportedOperationException(); }
4    public ConsList tail() { throw new UnsupportedOperationException(); }
5
6    public boolean isEmpty() { return true; }
7    public int size() { return 0; }
8
9    public int hashCode() { return 0; }
10   public boolean equals(Object that) { return this == that; }
11 };
```

Listing 1.12: Implementation of `EmptyCell` case of a `ConsList`.

Hash-Consing. The hash-consing [Got74] tactic is an optimization strategy that is inseparably connected with persistent data structures. It entails that selected objects that are equal are not present in memory more than once at a given point in time.

There are two main expected benefits of hash-consing: avoiding all redundancy by eliminating clones in memory, and the ability to use constant time reference comparisons instead of deep `equals` checks that are in $O(\textit{size of object graph})$. This is because hash-consing enforces the following invariant among selected objects: $\forall$ *objects* $x, y : x.equals(y) \Leftrightarrow x == y$, which allows any call to `equals` on shared objects to be replaced with a reference comparison.

Essentially, the hash-consing strategy memoizes [McC60; Mic68; Nor91] the equality relation between objects [Vaz+07]. For the `EMPTY_LIST` constant of Listing 1.12, we manually enforced that there exists only a single empty list, and therefore it makes use of reference equality (cf. Line 10) instead of invoking `equals`.

To extend the memoization of arbitrary `ConsList` instances (cf. Listing 1.14), a global cache is used to administrate the current universe of live objects, against which every new list is tested. The overhead introduced by hash-consing is the administration of the cache, and additional calls to `hashcode` and `equals` methods due to cache lookups. The expected benefits are the elimination of duplicate objects and constant time references comparisons in contrast to deep recursive `equals` calls.

In practice, maximal sharing can effectively improve the performance of programs that operate on immutable and persistent data structures. In the context of term rewriting engines van den Brand and Klint [vdBK07] observed memory savings of at least 50 % and up to 98.50 %. The authors reported that the increased term construction time was *"more than compensated for by fast equality checking and less use of space"*. However, depending on the scenario, hash-consing could also increase a program's execution time and memory consumption in absence of enough redundancy, due to the overhead of the global cache.

```
private static final class ConsCell<E> extends ConsList<E> {

  private final E head;
  private final ConsList<E> tail;

  ConsCell(E head, ConsList<E> tail) { this.head = head; this.tail = tail; }

  public E head() { return head; }

  public ConsList<E> tail() { return tail; }

  public boolean isEmpty() { return false; }

  public int size() { return 1 + tail.size(); }

  public int hashCode() { return Objects.hashCode(head) + 31 * tail.hashCode(); }

  public boolean equals(Object other) {
    if (other == this) { return true; }
    if (other == null) { return false; }
    if (getClass() != other.getClass()) { return false; }

    ConsCell<?> that = (ConsCell<?>) other;
    return Objects.equals(head, that.head) && Objects.equals(tail, that.tail);
  }
}
```

Listing 1.13: Implementation of `ConsCell` case of a `ConsList`.

In general, hash-consing avoids the allocation of duplicate objects and makes singleton instances available across otherwise independent parts of a program. To guarantee safe sharing, and not to introduce unexpected side-effects, object-to-be shared are required to be immutable or persistent.

Introducing hash-consing to a library requires cross-cutting and invasive changes to the library's code base. The following list sketches a minimal set of changes necessary to introduce hash-consing, without considering optimizations:

- replacing object allocations by calls to factory methods with cache lookups;
- maintaining a set of live objects in a thread-safe/concurrent object pool;
- replacing calls to `equals` by $==$, or implementing object identity inside `equals`;
- adding a new shallow equality comparator to objects that compares nested sub-structures with $==$ (because DAG sub-structures are already shared).

Hash-consing is an essential optimization strategy to consider for immutable and persistent data structures, therefore we raise the question if it possible to assess the applicability of this strategy to existing data structure libraries:

> **Research Question:** Hash-consing is an essential optimization strategy to consider for persistent data structures, which involves cross-cutting and invasive changes to a data structure library code base. How can we evaluate a priori the performance impact of applying hash-consing to a data structure library? *Research Question (RQ-6). Addressed by Chapter 6.*

Hash-Array Mapped Tries. The state-of-the-art data structure for persistent hash-sets and hash-maps is the Hash-Array Mapped Trie (HAMT) [Bag01]. HAMTs are wide and shallow search trees that encode the hash codes of the elements contained in the data structure in a prefix tree.

The first persistent implementation of HAMTs can be attributed [BR11] to Rich Hickey, the lead developer of the Clojure programming language. Collections that are implemented as persistent HAMTs are contained in standard libraries of JVM languages such as Clojure, Rascal, or Scala —that promote either functional or hybrid object-oriented and functional programming paradigms— and other widely-used programming languages such as Erlang, JavaScript, Haskell, and Ruby.

HAMTs are the de-facto standard for efficient persistent hash-set and hash-map data types that support sequential and parallel processing. HAMTs also share a common foundation with derived data structures for sequences [Stu+15; BR11] and concurrent processing [Pro+12]. We analyze the current state-of-the-art of HAMT implementations on the JVM, in order to improve the common core abstractions that influence the performance of all of the former mentioned data structures.

> **Research Question:** What are the strength and weaknesses of current state-of-the-art persistent data structures that are based on HAMTs, i.e., hash-sets and hash-maps? *Research Question (RQ-2). Addressed by Chapter 3.*

1.5 Platform-Specific Data Locality Challenges

Theoretical results from data structure research usually describe worst case, asymptotic and amortized complexities of operation runtimes and memory footprints of data structures, but do usually not consider non-functional properties such as throughput and response-time, expected memory footprint and expected run-time. The former list of important quality factors are not considered by non-parametric,

```
private static final class ConsCell<E> extends ConsList<E> {

  private final int hash;
  private final E head;
  private final ConsList<E> tail;

  private ConsCell(E head, ConsList<E> tail) {
    this.head = head; this.tail = tail;
    this.hash = Objects.hashCode(head) + 31 * tail.hashCode();
  }

  private static final Map<ConsList, ConsList> CACHE = new
      WeakShallowEqualsHashMap<>();

  public final static <E> ConsList<E> of(E head, ConsList<E> tail) {
    ConsList tmp = new ConsList(item, this);
    ConsList cached = CACHE.get(tmp);

    if (cached != null) {
      return cached;
    } else {
      cache.put(tmp, tmp);
      return tmp;
    }
  }

  public E head() { return head; }

  public ConsList<E> tail() { return tail; }

  public boolean isEmpty() { return false; }

  public int size() { return 1 + tail.size(); }

  public int hashCode() { return hash; }

  public boolean equals(Object other) { return this == other; }

  public boolean shallowEquals(Object other) {
    if (getClass() != other.getClass()) { return false; }

    ConsCell<?> that = (ConsCell<?>) other;
    return Objects.equals(head, that.head) && tail == that.tail;
  }
}
```

Listing 1.14: An extended `ConsCell` that supports hash-consing.

asymptotical algorithmic complexity and data-structure theory. As a consequence, it remains difficult to estimate the expected performance of collections on a specific platform. Data structures in Java and the JVM are in particular challenging to optimize, due to automatic memory management and garbage collection.

Optimizations for Tree Data Structures

HAMTs inherently feature data locality issues caused by their tree-based nature, notably when compared to mutable array-based data structures such as dense hashtables (which use consecutive arrays). The JVM disallows fine grained control over an object graph's memory layout. Instead the JVM hides details of memory allocations and de-allocations by automatically managing memory at a runtime-level. Therefore HAMTs present an optimization challenge on the JVM:

- The JVM currently does not allow custom memory layouts. A HAMT forms a prefix tree that consists of linked nodes. Each HAMT node is an object that could be arbitrarily positioned in memory, resulting in an increased number of cache misses.
- Arrays are objects too on the JVM. HAMT nodes use arrays to compactly store references to sub-trees. Hence, arrays introduce here yet another level of memory indirections, reducing data locality within nodes.

In contrast, system programming languages, such as C/C++, allow compact custom memory layouts that are tailored to the shape of the data, with possibly better data locality than on the JVM. We are interested in optimizing the memory footprint of HAMT data structures on the JVM.

Research Question: HAMT data structures inherently feature many memory indirections due to their tree-based nature. How can we optimize the memory footprint of such data structures on the JVM? *Research Question (RQ-3). Addressed by Chapters 4 and 5.*

Optimizations for Composite Data Structures

Collection libraries contain a small set of versatile and often used data structures. To avoid code duplication, standard libraries of languages such as Clojure, Java, and Scala avoid specialized data structures, if possible. Instead those libraries try to implement data structures by reusing and composing other collections. Two examples illustrate that philosophy. First, Clojure contains a persistent hash-map implementation, but Clojure's hash-set data structure is implemented as a wrapper

```
GenericBox<Integer> boxedItem = new GenericBox<>(13);
int item = boxedItem.get(); /* no cast required, however, performance
      degradations may occur due to automatic conversion of int to Integer */
```

Listing 1.15: Illustrating current limitations of primitive generics.

of a hash-map, mapping elements to boolean objects. Second, instead of a specialized multi-map,* Clojure and Scala contain multi-maps that nest sets as the values of a polymorphic map. While a specialized multi-map implementation would be more memory efficient, a composition solution saves collection library maintainers from duplicating an almost identical implementation. Note, that persistent data structure implementations are highly detailed and error prone, reuse by composition is thus a factor of quality control when data structures are written by hand.

In a setting where data structures are engineered by hand, reuse by composition is a trade-off, giving away performance in favor of a smaller code base with better maintainability and quality. We want to break the trade-off between reuse and variability by using generative programming [McI68; Big98; CE00].

By generating collection data structures (cf. Section 1.2), we expect that it is possible to generate specialized and more efficient data structures. Automation facilitates repeatability and a high level of quality of the generated results.

For the case of multi-maps, we are interested to overcome the limitations of composite implementations, improving drastically on the memory footprint without loss of storage, lookup and iteration efficiency. Achieving this goal requires the extension of traditional persistent data structures to accommodate for the abstractions of composite data structures. Furthermore, proper support on the level of data structures is a precursor to generate efficient specialized multi-maps.

Research Question (RQ-4): How can HAMTs be utilized (or extended) to efficiently support the implementation of specialized composite data structures such as multi-maps, avoiding the memory overhead of nesting existing collections into each other? *Addressed by Chapter 5.*

Optimizations based on Data Structure Content

In Section 1.1 we discussed the role of parametric polymorphism and generics in the context of collection libraries. The current version of the Java programming language, Java SE 8, restricts generics to reference types, i.e., sub-types of `java.lang.Object`.

*A multi-map is a data structure which 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.

Listing 1.15, that uses the `GenericBox` data type of Listing 1.6, illustrates the potential performance pitfalls when using generic data types with primitives. When substituting type parameters with a primitive type —such as **`byte`**, **`char`**, **`int`** or **`long`**— Java will apply *boxing*,[†] an automatic conversion of a primitive type to an equivalent object representation (`Byte`, `Char`, `Integer` or `Long`). Boxing of primitive values degrades performance, causing more cache misses and increasing the memory footprint of the program. Consequently, manual code specialization, as shown in Listing 1.5, is still required when top performance is required for processing data of primitive types.

The differentiation between primitive types and reference types in the Java programming language also affects the design of collections. Because of the above mentioned shortcomings of collections, when used with primitive data types, third party libraries were created that are specialized for primitive values, however at the expense of not supporting the storage of reference types along primitives values.

Research Question: Can we bring the advantages in efficiency of primitive collections to generic collections? E.g., how can we optimize data structures for numeric or primitive data types? Or, how can we mix elements of primitive types and reference types without boxing? *Research Question (RQ-5). Addressed by Chapter 5.*

1.6 Contributions

The remainder of this section elaborates on the origins and contributions of the forthcoming thesis chapters. Bibliography references are listed for all published articles. Furthermore, the list of research questions, and their mapping to forthcomings thesis chapters, is aggregated in Table 1.1.

For all published papers, the author of this thesis was also the primary author of the paper, the researcher who designed and executed the research method, and the programmer who build the contributing software.

Chapter 2. Towards a Software Product Line of Trie-Based Collections.

> *Michael J. Steindorfer and Jurgen J. Vinju. 2016. Towards a Software Product Line of Trie-Based Collections. In Proceedings of the 2016 International Conference on Generative Programming: Concepts and Experiences (GPCE 2016). ACM, New York, NY, USA. URL http://dx.doi.org/10.1145/2993236.2993251*

Chapter 2 discusses a product line for collection data structures that would relieve library designers from optimizing for the general case, and allows evolving the

[†] `http://docs.oracle.com/javase/specs/jls/se8/html/jls-5.html#jls-5.1.7`

Table 1.1: Summary of research questions and their mapping to chapters.

RQ-1	What are the commonalties and differences between different data structures in the domain of collection libraries? Can we generalize over the sub-domain of immutable collections by modeling variability and documenting implementation choices? *Addressed by Chapter 2.*
RQ-2	What are the strength and weaknesses of current state-of-the-art persistent data structures that are based on HAMTs, i.e., hash-sets and hash-maps? *Addressed by Chapter 3.*
RQ-3	HAMT data structures inherently feature many memory indirections due to their tree-based nature. How can we optimize the memory footprint of such data structures on the JVM? *Addressed by Chapters 4 and 5.*
RQ-4	How can HAMTs be utilized (or extended) to efficiently support the implementation of specialized composite data structures such as multi-maps, avoiding the memory overhead of nesting existing collections into each other? *Addressed by Chapter 5.*
RQ-5	Can we bring the advantages in efficiency of primitive collections to generic collections? E.g., how can we optimize data structures for numeric or primitive data types? Or, how can we mix elements of primitive types and reference types without boxing? *Addressed by Chapter 5.*
RQ-6	Hash-consing is an essential optimization strategy to consider for persistent data structures, which involves cross-cutting and invasive changes to a data structure library code base. How can we evaluate a priori the performance impact of applying hash-consing to a data structure library? *Addressed by Chapter 6.*

potentially large code base of a collection family efficiently. We present a small core of intermediate abstractions for immutable collection data structures that concisely covers a wide range of variations that feature class-leading performance.

Chapter 3. The CHAMP Encoding.

> *Michael J. Steindorfer and Jurgen J. Vinju. 2015. Optimizing Hash-array Mapped Tries for Fast and Lean Immutable JVM Collections. In Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2015). ACM, New York, NY, USA. URL http://dx.doi.org/10.1145/2814270.2814312*
>
> The paper is accompanied by a source code and benchmarking artifact* that passed the peer-reviewed OOPSLA 2015 Artifact Evaluation.
>
> Our results were picked up and independently replicated by industry.†

*`http://michael.steindorfer.name/publications/oopsla15-artifact`

†`https://bendyworks.com/leveling-clojures-hash-maps/`

The standard libraries of recent Java Virtual Machine languages, such as Clojure or Scala, contain scalable and well-performing immutable collection data structures that are implemented as HAMTs. In Chapter 3, we propose the Compressed Hash-Array Mapped Prefix-tree (CHAMP) encoding, that improves the overall performance of immutable sets and maps. The resulting general purpose design increases cache locality and features a canonical representation. It outperforms Scala's and Clojure's data structure implementations in terms of memory footprint and runtime efficiency of iteration (1.3–6.7 x) and equality checking (3–25.4 x).

Chapter 4. Specialization for Memory Efficiency.

> *Michael J. Steindorfer and Jurgen J. Vinju. 2014. Code Specialization for Memory Efficient Hash Tries (Short Paper). In Proceedings of the 2014 International Conference on Generative Programming: Concepts and Experiences (GPCE 2014). ACM, New York, NY, USA. URL http://dx.doi.org/10.1145/2658761.2658763*

In Chapter 4, we apply generative programming techniques to specialize HAMT nodes. We discuss how to reduce the number of specializations from a large exponential number to a small subset while still maximizing memory savings. With this techniques we achieved a median decrease of 55 % in memory footprint for maps and 78 % for sets compared to a non-specialized version, but at the cost of 20–40 % runtime overhead of the lookup operation.

Chapter 5. A Flexible Encoding for Heterogeneous Data was submitted to the 2016 edition of the *International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. A pre-print of the article is available:

> *Michael J. Steindorfer and Jurgen J. Vinju. 2016. Fast and Lean Immutable Multi-Maps on the JVM based on Heterogeneous Hash-Array Mapped Tries. arXiv:1608.01036. URL https://arxiv.org/abs/1608.01036*

In Chapter 5, we propose a general framework for Hash-Array Mapped Tries on the JVM which can store type-heterogeneous keys and values: a Heterogeneous Hash-Array Mapped Trie (HHAMT). Among other applications, this allows for a highly efficient multi-map encoding by (a) not reserving space for empty value sets and (b) inlining the values of singleton sets while maintaining a (c) type-safe API. We detail the necessary encoding and optimizations to mitigate the overhead of storing and retrieving heterogeneous data in a hash-trie. The new encoding brings the per key-value storage overhead down by a 2 x improvement. With additional inlining of primitive values it reaches a 4 x improvement.

Chapter 6. Performance Modeling of Maximal Sharing.

> *Michael J. Steindorfer and Jurgen J. Vinju. 2016. Performance Modeling of Maximal Sharing. In Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering (ICPE '16). ACM, New York, NY, USA. URL http://dx.doi.org/10.1145/2851553.2851566*
>
> The paper was awarded with the *Best Paper Award* of the *Industry and Experience* track.

Chapter 6 discusses an a priori performance modeling technique for advanced cross-cutting optimization strategies that depend on immutable data structures. We specifically model the potential performance implications of introducing "maximal sharing" (a.k.a. "hash-consing") to a library that does not implement maximal sharing yet. Maximal sharing may have great benefit in terms of time and space, or may have detrimental overhead, depending on the redundancy of data and the use of equality. With comparatively low effort we can predict the impact of maximal sharing on existing code upfront, and uncover optimization opportunities that otherwise would remain hidden.

1.7 Software Artifacts

In the scope of this thesis, several open-source software projects were created that are licensed under the *"Simplified" BSD 2-Clause License*,* facilitating the transfer of our research results within the scientific and open-source community, and the transfer of our results to industry.

Our projects either directly contain data structure implementations (Capsule), facilitate cross-language and cross-library benchmarking (Criterion), or support the generation and specialization of efficient immutable data structures (DSCG).

Capsule: The Hash Trie Collections Library.† Capsule is a collections library for the JVM and written in Java that contains implementations of our research results on persistent trie-based unordered data structures. In contrast to other persistent collection libraries on the JVM —that are either forks or ports of Clojure's and Scala's data structures— Capsule is engineered from ground up for Java. Furthermore, Capsule powers the built-in data structures of the Rascal programming language in production since 2014.

*`https://opensource.org/licenses/BSD-2-Clause`

†`https://michael.steindorfer.name/projects/capsule`

Criterion: The Benchmarking Suite for Immutable Collections across the JVM.[‡] Criterion is a benchmarking library that facilitates cross-language comparisons of persistent data structures, supporting Scala's immutable collections, Clojure's built-in data structures, and several collections libraries in the Java ecosystem. To our knowledge, Criterion is the first elaborate benchmarking attempt to benchmark immutable collection data structures across different library and language barriers.

DSCG: A Domain Specific / Data Structure Code Generator for Collections.[§] The Data Structure Code Generator (DSCG) is an ongoing effort to encapsulate the commonalities and differences in abstractions and features of trie-based collections. It aims to simplify the co-evolution of a product family of efficient data structures by applying generative programming techniques.

1.8 Further Reading

This concludes the summary of objectives and contributions of our research. Readers that are interested in contributions per field or subject, in particular, can find recommendations below:

Data Structure Design: Our contributions are language independent for the most part (and therefore portable), but also detail optimizations specific to the JVM.

Chapter 3 describes the CHAMP encoding, a successor to the state-of-the-art approach of HAMTs. We detail design choices and differences to Clojure's and Scala's immutable collections.

Chapter 5 details HHAMT that is a generalization of CHAMP for storing type-heterogeneous payloads. Amongst other applications, HHAMT by design unifies the long-standing divide between generic collections and primitive specialized collections. This topic might be of interested to Virtual Machine and core library engineers.

Code Generation: Chapter 2 describes our overall approach to use generative programming techniques for obtaining efficient trie-based collections, and discusses core intermediate abstractions.

Furthermore, Chapter 4 and Section 5.3 cover generative memory layout optimizations for trie-based data structures. The former details a pragmatic approach on a library level, the latter discusses a more sophisticated approach that works best with Virtual Machine (VM) support.

[‡] `https://michael.steindorfer.name/projects/criterion`
[§] `https://michael.steindorfer.name/projects/dscg`

Applications and Performance Modeling: Chapter 6 zooms out and focuses on library and application level optimizations that are applicable to immutable data structures. The chapter discusses profiling and performance modeling techniques to a priori investigate optimization opportunities.

Readers that are generally interested to learn more about immutable collections, advanced optimizations, and applications are suggested to progress in linear order.

Chapter 2

Towards a Software Product Line of Trie-Based Collections

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.

This chapter is based on the following published article: Michael J. Steindorfer and Jurgen J. Vinju. 2016. Towards a Software Product Line of Trie-Based Collections. In Proceedings of the 2016 International Conference on Generative Programming: Concepts and Experiences (GPCE 2016). ACM, New York, NY, USA. URL http://dx.doi.org/10.1145/2993236.2993251.

2.1 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* and Clojure†. Immutability allows optimizations that exploit the fact that data does not change [Hel15; SV16b], 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.

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.

*https://scala-lang.org
†https://clojure.org

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 [McI68; Big98; CE00]. 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 chapter 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.

2.2 Related Work

Software Product Lines and Dynamic Adaptation. We take a static Software Product Line (SPL) [CN01] perspective on collections to enable software reuse. Features of collections and variability are typically known at design time. Dynamic Software Product Lines [Hal+08] and Run-Time Adaptation [Alv+09] 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 [SSS79]. On the Java Virtual Machine (JVM), Shacham et al. [SVY09] 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 [SB97].

Hawkins et al. worked on declarative and provable specifications and synthesis of data structures with complex sharing, both for the sequential [Haw+11] and concurrent [Haw+12] case.

Loncaric et al. [LTE16] 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 chapter does.

Specializing for Primitive Data Types. Ureche et al. [UTO13] 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 [dlBri59] and named a year later by Fredkin [Fre60]. An AMT [Bir77; Bag00] 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) [Bag01] 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 [Dri+86; Oka99]. In previous work we introduced the Compressed Hash-Array Mapped

Prefix-tree (CHAMP) [SV15], 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 [BR11] published a technical report about efficient immutable vectors that improved runtimes of split and merge operations to a logarithmic bound. Stucki et al. [Stu+15] improved upon the latter and added a broad scale evaluation.

Concurrent HAMTs. Prokopec et al. [Pro+12] 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.

2.3 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 [vDK02]. The full model has been archived [Ste16] and details the variability in the domain of collections, making commonalities and differences of configurations, constraints among them, explicit.

A `trie` (cf. Listing 2.1) 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, we store a copy at the accept nodes to cater for possible hash collisions. The features `ChunkUnit`, `ChunkLength` and `EncodingDirection` determine

```
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)
```

Listing 2.1: Extract of a feature model describing trie data structure encodings.

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,

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

Listing 2.2: ADT term for the partitioning of a set of family members called CHAMP, parametrized by a size bound (i.e. 32).

```
list[PartitionCopy] transform(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 2.3: Linearization and transformation from domain specific `copyAndInsert` primitive to intermediate abstraction.

a feature model allows for customization towards specific workloads (e.g., sparse vectors). For efficiency trade-offs it is important to distinguish between HAMTs that store `dataAsLeafs` and HAMTs that allow for `mixedNodes` internally [SV15].

We currently generate unordered set, map, and multi-map data structures based on the state-of-the-art HAMT variants: HAMT [Bag01], CHAMP [SV15], and HHAMT [SV16a]. 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.* In future work we plan supporting vectors and concurrency.

2.4 Intermediate Generator Abstractions

We use a form of these feature models to configure the Data Structure Code Generator (DSCG) that actually implements each variant.* The DSCG is implemented in Rascal, a Domain-Specific Language (DSL) designed for analyzing, processing, transforming and generating source code [KvdSV09]. 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.

*https://michael.steindorfer.name/projects/capsule/
*https://michael.steindorfer.name/projects/dscg/

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 [SV15]. 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 [SV15].

Listing 2.2 shows the partition configuration of a hash-map encoded in CHAMP [SV15]. 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 [SV14; SV16a].

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 Listing 2.4.

By transforming 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 2.3. A linearized view 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 (cf. Listing 2.5).

```
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));
```

Listing 2.4: Generated (naive) Java code for `copyAndInsert` that is derived from a logical partition layout.

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

Listing 2.5: Generated (optimized) Java code for `copyAndInsert` that is derived after linearization of the partition layout and fusion of `rangeCopy` operations.

2.5 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 [SV16a]. The generated code is efficient, overall outperforming competitive state-of-the-art collections [SV15; SV16a], and —when specialized for primitive data types— they match the memory footprints of best-of-breed primitive collections [SV16a].

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.

Chapter 3

The CHAMP Encoding

The data structures under-pinning collection API *(e.g. lists, sets, maps) in the standard libraries of programming languages are used intensively in many applications.*

The standard libraries of recent Java Virtual Machine languages, such as Clojure or Scala, contain scalable and well-performing immutable collection data structures that are implemented as Hash-Array Mapped Tries (HAMTs). HAMTs already feature efficient lookup, insert, and delete operations, however due to their tree-based nature their memory footprints and the runtime performance of iteration and equality checking lag behind array-based counterparts. This particularly prohibits their application in programs processing larger data sets.

*In this chapter, we propose changes to the HAMT data structure that increase the overall performance of immutable sets and maps. The resulting general purpose design increases cache locality and features a canonical representation. It outperforms Scala's and Clojure's data structure implementations in terms of memory footprint and runtime efficiency of iteration (*1.3–6.7 x*) and equality checking (*3–25.4 x*).*

This chapter is based on the following published article: Michael J. Steindorfer and Jurgen J. Vinju. 2015. Optimizing Hash-array Mapped Tries for Fast and Lean Immutable JVM Collections. In Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2015). ACM, New York, NY, USA. URL http://dx.doi.org/10.1145/2814270.2814312.

3.1 Introduction

In this chapter we reduce memory overhead and runtime performance overhead from the implementations of immutable collections on the Java Virtual Machine (JVM). Collections under-pin many (if not most) applications in general purpose programming languages as well as domain specific languages that are compiled to the JVM. Optimizing collections implies optimizing many applications.

Immutable collections are a specific area most relevant to functional/object-oriented programming such as practiced by Scala* and Clojure† programmers. With the advance of functional language constructs in Java 8 and functional APIs such as the stream processing API [Bib+15], immutable collections become more relevant to Java as well. Immutability for collections has a number of benefits: it implies referential transparency without giving up on sharing data [IV02]; it satisfies safety requirements for having co-variant sub-types [IV02]; it allows to safely share data in presence of concurrency.

The prime candidate data structure for efficient immutable collections is the Hash-Array Mapped Trie (HAMT) by Bagwell [Bag01]. This data structure design was already ported from C++ to the JVM platform and is used especially in the Scala and Clojure communities. However, these ports do not perform optimally, because the performance characteristics of the JVM are substantially different from the performance characteristics of C/C++ runtimes. The unique properties of the JVM have not sufficiently been exploited yet. The following properties hinder efficiency of data structures that are ported directly from C/C++ to the JVM:

- The JVM currently does not allow custom memory layouts. A HAMT forms a prefix tree and therefore consists of linked nodes. Each HAMT node is represented as an object that is arbitrarily positioned in memory, resulting in an increased number of cache misses.

- Arrays are objects too. HAMT nodes use arrays to compactly store references to sub-trees. Hence, arrays add here yet another level of memory indirections.

In contrast, C/C++ allows custom memory layouts that are tailored to the shape of the data, without the need of memory indirections. With possibly better data locality than on the JVM, C/C++ runtimes directly place the content of statically sized arrays into an object's or struct's memory region.

Fine-tuning data structures for cache locality usually improves their runtime performance [LFN02]. However, HAMTs inherently feature many memory indirections due to their tree-based nature, notably when compared to array-based data structures such as hashtables. Hence HAMTs are challenging to optimize on the JVM.

*http://scala-lang.org
†http://clojure.org

Our goal is to optimize HAMT-based data structures such that they become a strong competitor of their optimized array-based counterparts in terms of speed and memory footprints. Our contributions can be summarized as follows:

- We introduce a new generic HAMT encoding, named Compressed Hash-Array Mapped Prefix-tree (CHAMP). CHAMP maintains the lookup, insertion, and deletion runtime performance of a HAMT, while reducing memory footprints and significantly improving iteration and equality checks.
- In relation to CHAMP, we discuss the design and engineering trade-offs of two wide-spread HAMT implementations on the JVM that are part of Clojure's and Scala's standard libraries.
- We evaluate the memory footprints and runtime performance of CHAMP, compared to immutable sets and maps from the aforementioned two libraries.

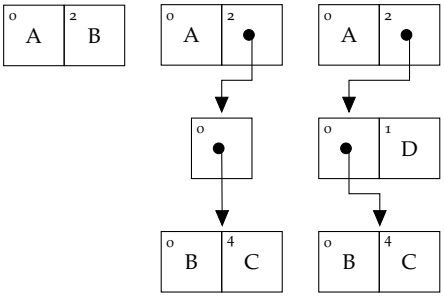
Speedups of CHAMP range between 9.9 x and 28.1 x, for an example program analysis algorithm that uses classical set and relational calculus in a fixed-point loop. In micro-benchmarks, CHAMP reduces the memory footprint of maps by 64 %, and of sets by 52 % compared to Scala. Compared to Clojure, CHAMP reduces the memory footprint by 15 % for maps, and of 31 % for sets. Compared to both, iteration speeds up 1.3–6.7 x, and equality checking improves 3–25.4 x. All speedups are medians.

Roadmap. The chapter is structured as follows:

- Section 3.2 discusses background and optimization opportunities for HAMTs.
- Section 3.3 describes the foundations of CHAMP: the differently ordered data layout of the tree nodes, and compression via an additional bitmap.
- Section 3.4 introduces an efficient algorithm to canonicalize HAMTs upon deletion, and details how to use this to improve equality checking.
- Section 3.5 discusses design and implementation trade-offs that are related to our core contributions and necessary to discuss the evaluation results.
- Section 3.6 compares CHAMP against Clojure's and Scala's data structures in terms of memory footprint and runtime efficiency with microbenchmarks.
- Section 3.7 extends the former benchmarks to a realistic use-case and workload.
- Section 3.8 discusses threats to validity and implementation differences.
- Section 3.9 addresses related work, before we conclude in Section 3.10.

All source code discussed, of data structures and benchmarks, is available online.[‡]

[‡] `http://michael.steindorfer.name/papers/oopsla15-artifact`



(a) A and B (b) C (c) D

$$hash(A) = 32_{10} = 0\,1\,0\ldots_{32}$$
$$hash(B) = 2_{10} = 2\,0\,0\ldots_{32}$$
$$hash(C) = 4098_{10} = 2\,0\,4\ldots_{32}$$
$$hash(D) = 34_{10} = 2\,1\,0\ldots_{32}$$

(d) hash codes in base 32

Figure 3.2: Inserting three integers into a HAMT-based set (3.1(a), 3.1(b), and 3.1(c)), on basis of their hashes (3.1(d)).

3.2 Background

Although sets and maps need no motivation, immutable collections are not as commonplace in the object-oriented domain yet. We enumerate several applications here to convince the reader of the interest in optimizing them.

A number of (semi-)formal mechanisms for analyzing sets and relations, based on Codd's relational calculus or Tarski's relational algebra have or can have immutable collections under-the-hood. Example Java-based systems in this category are JRelCal [Rad08], JGraLab [Ebe+07] and Rascal [KvdSV09]. Implementations of logic programming formalisms like, Datalog [HVdM06], can also benefit from optimizations of these data-structures [Ram+95]. Both categories of formalisms are used actively in static analysis of object-oriented programs. Another application domain is the representation of graphs and models in model-driven engineering.

Making immutable collections efficiently available in object-oriented languages transfers the benefits of the aforementioned formal languages to mainstream programming: immutability enables equational reasoning.

HAMTs Compared to Array-based Hashtables

A general trie [dlBri59; Fre60] is a lookup structure for finite strings that acts like a Deterministic Finite Automaton (DFA) without any loops: the transitions are the characters of the strings, the internal nodes encode prefix sharing, and the accept nodes may point to values associated with the strings. In a HAMT, the strings are the bits of the hash codes of the elements stored in the trie. Depending on the branching factor we may use different sizes of chunks from the hash code for indexing into the (sparse) array of child nodes.

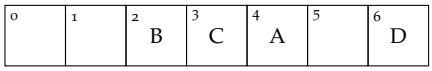


Figure 3.3: This figure shows a collision-free array-based hash set —with prime number table size 7 and load factor of 75 %— that is equivalent to Figure 3.1(c).

Figures 3.1(a), 3.1(b), and 3.1(c) graphically illustrate the structure of a small HAMT set with branching factor 32 after step-by-step inserting the objects A, B, C, and D. HAMTs encode the hash codes of the elements in their tree structure (cf. Figure 3.1(d)). The prefix tree structure grows lazily upon insertion until the new element can be distinguished unambiguously from all other elements by its hash code prefix. The index numbers in the left top corner of each node refer to the positions of elements in an imaginary sparse array. This array is actually implemented as a 32-bit bitmap and a completely filled array with length equal to the node's arity. To change a HAMT set into a map, a common method is to double the size of the array and store references to each value next to each key.

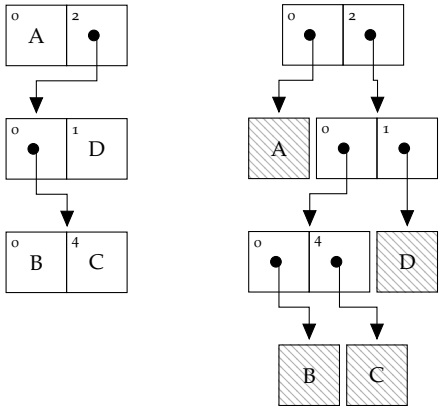
Figure 3.3 illustrates the same data stored in a more commonly known data structure, an array-based hashtable with table size 7 and load factor of 75 %. The buckets assigned to elements are calculated by hashcode mod 7. Comparing these two figures we highlight the following inherent drawbacks of HAMTs against array-based hashtables:

Memory overhead: Each internal trie node adds an overhead over a direct array-based encoding, so finding a small representation for internal nodes is crucial. On the other hand, HAMTs do not need expensive table resizing and do not waste (much) space on `null` references.

Degeneration on delete: Any delete operations can cause a HAMT to deviate from its most compact representation, leading to superfluous internal nodes harming cache locality, adding indirections for lookup, insertion and deletion, and increasing memory size. Delete on most hashtable implementations is a lot less influential.

Non-locality slows down iteration: When iterating over all elements, a hashtable benefits from locality by linearly scanning through a continuous array. A HAMT, in comparison, must perform a depth-first in-order traversal over a complex object graph (going up and down), which increases the chance of cache misses.

Equality is punished twice: While iterating over the one HAMT and looking up elements in the other, the non-locality and a possibly degenerate structure make equality checking an expensive operation.



(a) Internal (b) Leaf

Figure 3.4: HAMT-based sets with values in internal nodes versus values at the leaves.

Mutable and Immutable Update Semantics

HAMTs are suitable to implement data structures with mutable and immutable update semantics. The two variants differ in how and when nodes have to be reallocated. Mutable HAMTs reallocate a node if and only if the node's arity changes [Bag01]. Otherwise, values or sub-node references are updated in-place without reallocation. In contrast to mutable HAMTs, immutable HAMTs perform path-copying on updates [ST86; Oka99]: the edited node and all its parent nodes are reallocated. The resulting new root node satisfies the immutability property by resembling the updated path-copied branch and, for the remainder, references to unmodified branches.

Memory Layouts and Hash Code Memoization

Figure 3.4 illustrates the two principle choices for a HAMT's memory layout: storing values next to sub-nodes directly in *internal* nodes, as opposed to storing them at the *leaf* level.

The former approach originates from Bagwell [Bag01], it increases locality and saves space over the latter. Because a HAMT encodes the prefixes of hash codes implicitly in its tree structure, it can avoid storing full 32-bit hash codes. As a result, this design yields a very low memory footprint at the potential cost of increased runtimes of update operations.

The latter approach stores elements in leaf nodes, separated from inner prefix tree nodes. While leaf nodes increase the data structure's memory footprint, they enable

```
abstract class HamtCollection {
  HamtNode root; int size;

  class HamtNode {
    int bitmap;
    Object[] contentArray;
  }
}
```

Listing 3.1: Skeleton of a HAMT in Java with internal values.

storage of additional information along the elements. Scala for example memoizes the hash codes of elements inside the leafs. Memoized hash codes consequently enable fast failing on negative lookups by first comparing hash codes, and avoid recalculation of hash codes upon prefix expansion.

CHAMP builds upon the internal nodes design and primary optimizes for smaller footprints and cache locality. Nevertheless, to reconcile both designs in our evaluation section, we will also analyze a CHAMP variant that supports hash code memoization in Section 3.5.

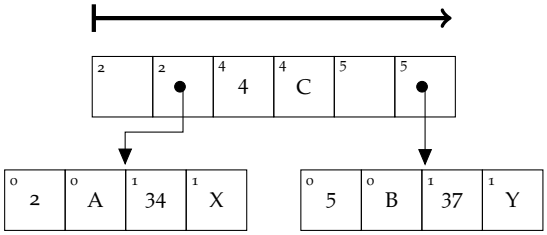
3.3 Data Locality

Listing 3.1 shows a Java source code skeleton that is the basis for implementing HAMT collections with the internal nodes design. Traditionally [Bag01], a single 32-bit integer bitmap is used to encode which slots in the untyped array are used, together with a mapping function that calculates offsets in the array by counting bits in the bitmap. For set data structures, we can use Java's `instanceof` operator to distinguish if an array slot holds either an element, or a sub-node reference. This encoding orders values by their hash codes—the bitmap compression solely eliminates empty slots.

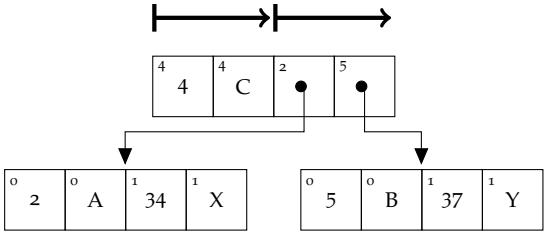
In the following, we first devise a new compact data layout for internal trie nodes which enables faster iteration and equality checking. The causes of the performance increase are compactness and locality, both leading to better cache performance, and avoiding the use of the `instanceof` operation.

Partitioning Values and Sub-Nodes

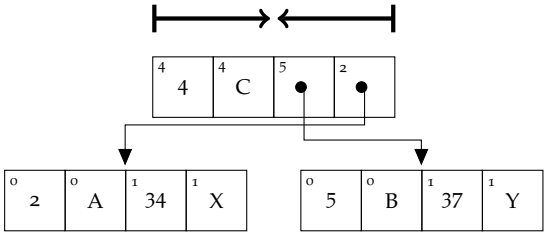
Maps with internal values (as opposed to sets) require additional storage room for the values. To cater for this the dense array can be allocated at twice the normal size such that next to each key a reference to a value may be stored. Each index is then multiplied by 2 to skip over the extra slots.



(a) Current HAMT ordering with 1-bit state.



(b) Conceptual CHAMP ordering with 2-bit state.



(c) Final CHAMP ordering with 2-bit state.

Figure 3.5: Various implementations of HAMT maps with values in internal nodes. The index numbers in the top left corners denote the logical indices for each key/-value entry and not their physical indices. Figure 3.5(a) depicts Clojure's HAMT implementation that indicates sub-nodes by leaving the array slot for the key empty.

Figure 3.5(a) exemplifies a HAMT-based map with internal values as found in Clojure. Accidentally, the root node has two child nodes, a few empty slots with `null` references and a key/value pair in the middle. Due to the fixed tuple length of two, an empty array slot is wasted per sub-node reference. Note that the C/C++ HAMT implementation of Bagwell [BR11] also uses fixed length tuples of size two for maps, however C/C++ offers better reuse (e.g., with union types) of the empty slots.

HAMT Design Results in Poor Iteration Performance. One aspect that is directly influenced by the order of array elements is iteration performance. The order of child nodes and values depends entirely on the data structure's content; sub-nodes and internal values may alternate arbitrarily. An in-order traversal for iteration will switch between the trie nodes a few times, because values and internal nodes alternate positions in the array, causing a bad cache performance. For iteration, the HAMT design requires to go through each node at most $m + n$ times, where n equals the HAMT's total number of sub-tree references and m equals the total number of references to internal data entries.

CHAMP Design Improves Iteration Performance due to a Cache-Oblivious Reordering with Compression. To increase data locality during iteration —and further to remove the need for empty array slots— we propose to split the single untyped array conceptually into two typed arrays (cf. Figure 3.5(b)): one array holding sub-node references, and another one holding values. This split effectively reorders the elements, while the ordering within each group remains. The element order is irrelevant anyway in already unordered set and map data structures. We remove the need for using `instanceof`, by introducing an extra bitmap that makes the separation between sub-nodes and internal values explicit. For iteration, the proposed CHAMP design reduces the complexity from $\mathcal{O}(m + n)$ to $\mathcal{O}(n)$. A full traversal in CHAMP requires exactly n node visits, because it can yield all internal values before descending for each sub-node exactly once.

Mitigating Memory Overhead. The CHAMP design conceptually requires two arrays and two accompanying bitmaps. A naive CHAMP implementation would introduce significant overhead compared to a HAMT. Figure 3.5(b) shows how we mitigate the incurred overhead by sharing one array for the two compressed array sequences. The second arrow indicates the need for the second bitmap. Two bitmaps are necessary to compute indices for either kind of value. We propose to use one 32-bit bitmap, called `datamap`, to store if a branch is either absent or a value reference, and another 32-bit bitmap, called `nodemap`, to store if a branch is either absent or a sub-node reference. Especially for maps we can spare the extra bitmap because we are saving an empty array cell for every sub-node due to our explicit encoding.

Mitigating Runtime Overhead

We have increased data locality at the cost of a more expensive index calculation that requires an extra bitmap and additional bit-level operations. This directly influences the runtime performance of lookup, insertion, and deletion. Each of these operations now requires a case distinction with, in worst case, two separate lookups in the two distinct bitmaps, to decide in which group an element is present. Because we compactly store both groups in a single array, we need to perform offset-based indexing when accessing the second group of sub-node references. Both the offset and the index calculations require more bit-level operations on the `datamap` and `nodemap`, as explained below.

Listing 3.2 illustrates Java snippets of how the indexing is usually implemented. Lines 1–3 shows the masking function that selects the prefix bits based on the node level in the tree (`shift == 5 * level`). The index function (lines 4–6) requires a `bitpos` variable with a single non-zero bit, designating one of the 32 possible branches. It then maps from the `bitmap/bitpos` tuple to a sparse-array index by counting the non-zero bits in `bitmap` on the right of `bitpos`.

Lines 8–18 illustrate indexing into a traditional HAMT that requires only a single call to `index`. For sets (`WIDTH = 1`), the content array does not have empty cells with `null`, for maps (`WIDTH = 2`) it follows the convention that an empty key slot indicates that the value slot is a sub-node reference.

Lines 20–34 show our proposal that requires different methods for accessing keys and sub-nodes. Accessing keys works equally to current HAMTs, however we call the `index` function with the `datamap`. For accessing sub-nodes we first calculate an `offset` (with `datamap`) before calling the `index` function with `nodemap`.

One option to remove the overhead of the offset calculation is to cache its value in a `byte` field. However, the extra `byte` pushes the memory layout of an internal node right over the JVM's 8-byte alignment edge, which seriously influences the memory footprint. Instead we remove the need for the additional field as follows. In our optimized and final CHAMP encoding, as displayed in Figure 3.5(c), we reverse the storage order of the nodes in the back of the array. We can then perform `contentArray.length - 1 - index` instead of the previous `offset + index` calculation. Since the `length` field is there anyway in the array we pay no extra cost in memory. CHAMP's optimized index calculation code that mitigates overhead is displayed in Listing 3.2, lines 36–42.

Implementing Fast Iterators

To support Java's `Iterable<T>` and `Iterator<T>` interfaces, our code is layered as in Listing 3.1. The outer (collection) layer provides key, value and entry iterators and a node iterator for the entire trie. The inner (trie node) layer provides separate

iterators for the internal values and for the child nodes of each internal node. As a result, the internal iterators essentially reflect a node's logical partitioning.

We implemented the node iterator using a stack interface. Since we statically know the maximal trie depth, the stack can be implemented in a pre-allocated cache-friendly array.

The idea of using pre-allocated stack iterators is not novel. Scala's HAMT implementations are already leveraging such iterators. Our own implementation is more elaborate to achieve the worst case iteration complexity reduction from $\mathcal{O}(m+n)$ to $\mathcal{O}(n)$ as discussed in earlier in Section 3.3.

Summary

We have increased locality by reshuffling the references in a trie node, at the cost of more bit-level arithmetic. Otherwise lookup, insertion, and deletion are unchanged. For iteration, the proposed CHAMP design reduces complexity from $\mathcal{O}(m+n)$ to $\mathcal{O}(n)$. For maps we avoid empty slots in the arrays and thus save memory. In the evaluation section we will show that the net result of our design is satisfying.

3.4 Canonicalization

Another way to increase locality and to further save memory is to keep a HAMT in a compact canonical representation, even after deleting elements. For example, in Figure 3.1(b) removing object C from the deepest node would yield a perfectly valid HAMT. However, in the optimal case, deletion would restore the state of Figure 3.1(a), resulting in a smaller footprint and less dereferencing upon lookup, insertion, and deletion.

Clojure's HAMT implementations do not compact on delete at all, whereas Scala's implementations do. In the remainder of this section we contribute a formalization (based on predicates and an invariant) that details how a HAMT with inline values can efficiently be kept in a compact canonical form when deleting elements. Bagwell's original version of insert is enough to keep the tree canonical for that operation. All other operations having an effect on the shape of the trie nodes can be expressed using insertion and deletion.

Contract for Canonicalization

We formalize the canonical form of internal trie nodes by a strict invariant for trie nodes. The reasons are that canonicalization depends on the children of trie nodes to be in canonical form already and the many operations on HAMTs are somewhat complex. An explicit invariant helps in implementing the canonical form correctly and in optimizing the code. We need two basic properties to express the invariant:

```
static final int mask(int hash, int shift) {
  return (hash >>> shift) & 0b11111;
}
static final int index(int bitmap, int bitpos) {
  return Integer.bitCount(bitmap & (bitpos - 1));
}

// HAMT indexing with 1-bit state
Object getKeyOrNode(K key, int hash, int shift) {
  int bitpos = 1 << mask(hash, shift);

  int index = WIDTH * index(this.bitmap, bitpos);
  if (contentArray[index] != null) {
    return contentArray[index];
  } else {
    contentArray[index + 1];
  }
}

// Proposed CHAMP indexing with 2-bit state
K getKey(K key, int hash, int shift) {
  int bitpos = 1 << mask(hash, shift);

  int index = WIDTH * index(this.datamap, bitpos);
  return (K) contentArray[index];
}

Node getNode(K key, int hash, int shift) {
  int bitpos = 1 << mask(hash, shift);

  int offset = WIDTH * Integer.bitCount(this.datamap);
  int index = offset + index(this.nodemap, bitpos);
  return (Node) contentArray[index];
}

// Optimized CHAMP indexing into sub-nodes
Node getNode(K key, int hash, int shift) {
  int bitpos = 1 << mask(hash, shift);

  int index = contentArray.length - 1 - index(this.nodemap, bitpos);
  return (Node) contentArray[index];
}
```

Listing 3.2: Index calculations for the various designs.

Arity (local): The arity of a node designates the number of outgoing edges. In CHAMP the `arity` equals the sum of `nodeArity` and `payloadArity`, counting bits set to 1 in `nodemap` and `datamap` respectively.

Branch Size (non-local): The branch size equals the total number of elements that are transitively reachable from a node. Later we will find an approximation for branch size which can be computed locally.

We assume both properties are available as methods on the objects of the internal nodes. The following class invariant asserts canonically minimal trie nodes:

CHAMP invariant:

$$\texttt{branchSize} \geq 2 * \texttt{nodeArity} + \texttt{payloadArity}$$

The invariant states that sub-trees with arity less than 2 are not allowed. This implies that single elements should always be inlined and singleton paths to sub-tries should be collapsed. The invariant holds for all nodes on all levels.

Deletion Algorithm

Deletion is a recursive operation on the tree. To satisfy the invariant, delete on a trie node structure should be the exact inverse operation of insert. Listing 3.3 contains a pseudocode description of the delete operation. For a given `key` the search starts at the root node. If the node contains the search key locally, the operation removes the data tuple from the node and returns an updated node. Otherwise, if the node contains a sub-tree for a given hash-prefix, the operation will descend recursively.

If the hash prefix is in `datamap` (line 2) and the value stored matches the key we are looking for then we can remove a value right here. In case a CHAMP instance consists of a single root node with a singe element, an `EMPTY_NODE` constant is returned, otherwise we return a copy of the node without the current element. We may temporarily generate a singleton value node (which is allowed by the invariant since it does not have a parent yet) but later in line 21 on the way back from the recursion this singleton will be collapsed. A singleton can only fall through, when delete collapses a multi-node tree of two elements to a tree with a single root node with a single element.

Note that both operations in line 21 (removal of `subNode` and inlining of `key`) need to be executed atomically to ensure the invariant. We provide a copying primitive (named `copyAndMigrateFromNodeToInline`) in our implementation that performs both modifications with a single array-copy.

If the hash prefix is in the `nodemap` (line 9) then delete is called recursively. The compaction to satisfy the invariant happens on the way back from the recursion, dispatching on the values of arity and branch size of the current node and the

received new node. If the current node's size is 1 then we may pass the new node to our own parent if it only contains values (line 17). If the received node has a bigger size, then we make a copy of the current singleton node with the new child at the right position (line 19). The final case is when the new child node wraps a value, generated by line 6, and we can inline it here right now. Line 23 implements the case where no compaction or inlining takes place at all.

Deletion Implementation Details. The object-oriented implementation of the delete algorithm involves careful consideration. It should avoid re-computations of hash codes, avoid allocation of temporary objects and it should maintain the class invariant. Also, computing the predicates used in the algorithm may become a bottleneck, especially computing the branch size of a trie node is problematic since this operation is linear in the size of the sub-tree. The following design elements are necessary to make the delete operation efficient:

- Passing a single state object through the recursive calls of the delete method, like a reference parameter, to record whether a modification was made.
- Inspecting the state of a received node by dispatching on the `arity` and `branchSize` properties. Using these properties we avoid the use of `instanceof`.
- Abstracting `branchSize` into an over-approximation which can be computed without recursion or caching. For the deletion algorithm (cf. Listing 3.3) we need to differentiate three states: if a sub-tree has no elements, exactly one element, or more. We can substitute calls to `branchSize` with calls to the `sizePredicate` method (cf. Listing 3.4) that returns a `byte` representation of the aforementioned three states.

Structural Equality Checking

Tree compaction on delete lays the groundwork for faster (structural) equality checking. In an ideal world without hash collisions we would short-circuit recursive equality checks without further ado: if two nodes that are reachable by the same prefix have different bitmaps it is guaranteed that their contents differ. Together with short-circuiting on equal references (then the sub-tries are guaranteed equal), the short-circuiting on unequal bitmaps makes equality checking a sub-linear operation in practice.* Only when the heap graphs of two equal HAMTs are disjunct, then a full linear traversal is necessary to assert equality.

*Note that an alternative implementation of equals —e.g., such as implemented in Clojure— is to iterate over one tree while looking up each key in the other until the first key which is not found. This operation is especially expensive on large HAMTs as it performs in $\mathcal{O}(n\ log_{32}(n))$.

```
delete(node: Node, key: Object): (Boolean, Node) {
  if (key in datamap) {
    if (node.arity == 1)
      return (true, EMPTY_NODE)
    else
      return (true, node without key)
  }

  if (∃subNode for which key is in nodemap) {
    (isModified, resultNode) = delete(subNode, key)

    if (isModified == false) // short-circuit
      return (false, node)

    if (node.arity == 1)
      if (resultNode.branchSize == 1) // propagate
        return (true, resultNode)
      else
        return (true, node updated with resultNode)
    else if (resultNode.branchSize == 1) // inline
      return (true, (node without subNode) with key)
    else
      return (true, node updated with resultNode)
  }

  return (false, node) // key not found
}
```

Listing 3.3: Pattern match logic for deletion in pseudocode.

```
byte sizePredicate() {
  if (this.nodeArity() == 0)
    switch (this.payloadArity()) {
      case 0: return SIZE_EMPTY;
      case 1: return SIZE_ONE;
      default:return SIZE_MORE_THAN_ONE;
    }
  else return SIZE_MORE_THAN_ONE;
}
```

Listing 3.4: `sizePredicate` method used for compaction.

Still, hash collisions do occur in practice. In a HAMT, a collision is detected when all hash code bits are consumed and two elements cannot be differentiated. Similar to a hashtable, a chained bucket can then be created at the bottom of the tree to hold both elements (or more). To be able to see why short-circuiting on unequal bitmaps is still possible, even though the structure of the buckets depends on dynamic insertion and deletion order, consider that we short-circuit only on hash codes prefixes being unequal and not on the actual values. This means that if the Java hash code contract —unequal hash codes imply unequal values— has been implemented correctly the short-circuiting is safe. Listings 3.5 and 3.6 show the source code of `equals` methods (regular node and hash collisions node) of a CHAMP-based set implementation.

Canonical representations enable equality checks to benefit from the persistent nature of immutable HAMTs. Like lookup, insert, and delete already did, equality checks can exploit short-circuiting due to incremental updates and sharing of intermediate nodes between different trie instances.

Summary

Compacting trie nodes on delete and only lazily expanding them on insert makes sure they are always in a canonical and compact state. The algorithm satisfies an invariant for trie nodes and needs to be implemented with care to mitigate the overhead. We save memory and gain locality, but we increase CPU overhead with more complex bitmap compaction. The evaluation in Section 3.6 analyses the true impact of the proposed trade-offs.

3.5 Memoization and Hash Codes

In the following we discuss design and performance implications of memoization of hash codes on two different levels: on element basis, and on collection level. For the latter case we discuss implications of incrementally updating them.

Memoizing Collection Hash Codes. A key design element of CHAMP is to use the outer wrapper objects to cache collection hash codes and to incrementally update these hash codes as elements are added or removed. This requires insertion-order independent and reversible hash code computations. Due to the JVM's 8-byte memory alignment, adding the hash code field to CHAMP does not increase its memory footprint.*

In contrast, memoization of collection hash codes is not an option for Scala, because in their class hierarchy every node is also a collection. Adding another field to each node would increase the memory footprint of inner nodes by 8 bytes.

*This is valid for JVM instances with less than 32GB heaps with the *CompressedOops* option enabled (default). `https://wikis.oracle.com/display/HotSpotInternals/CompressedOops`.

```
boolean equals(Object other) {
  if (other == this) return true;
  if (other == null) return false;
  if (getClass() != other.getClass())
    return false;

  ChampNode<?> that = (ChampNode<?>) other;

  if (datamap != that.datamap)
    return false;
  if (nodemap != that.nodemap)
    return false;

  if (!Arrays.equals(nodes, that.nodes))
    return false;

  return true;
}
```

Listing 3.5: `equals` method of a regular CHAMP node.

```
boolean equals(Object other) {
  if (other == this) return true;
  if (other == null) return false;
  if (getClass() != other.getClass())
    return false;

  HashCollisionNode<?> that = (HashCollisionNode<?>) other;

  if (hash != that.hash)
    return false;

  for (K key : keys)
    if (!that.contains(key))
      return false;

  return true;
}
```

Listing 3.6: `equals` method of a set's hash collision node.

Table 3.1: Worst case number of invocations of `hashCode`/`equals` per HAMT operation. The numbers exclude full hash collisions, but assume distinct hash codes with matching prefixes. For each operations we distinguish between the succeeding and the failing case. The table is split by the following HAMT features: memoizing and incrementally updating the collection hash code, and memoizing hashes of the keys.

Operation	**`hashCode` / `equals` Calls per Data Structure and Feature Set**							
	HAMT Map				**HAMT Set**			
	¬Incremental		**Incremental**		**¬Incremental**		**Incremental**	
	¬Memo	**Memo**	**¬Memo**	**Memo**	**¬Memo**	**Memo**	**¬Memo**	**Memo**
Lookup $_{(\neg\text{Contained})}$	1 / 1	1 / 0	1 / 1	1 / 0	1 / 1	1 / 0	1 / 1	1 / 0
Lookup $_{(\text{Contained})}$	1 / 1	1 / 1	1 / 1	1 / 1	1 / 1	1 / 1	1 / 1	1 / 1
Insert $_{(\neg\text{Contained})}$	2 / 1	1 / 1	3 / 1	2 / 0	2 / 1	1 / 0	2 / 1	1 / 0
Insert $_{(\text{Contained})}$	1 / 1	1 / 0	1 / 1	1 / 1	1 / 1	1 / 1	1 / 1	1 / 1
Replace $_{(\text{Contained})}$	1 / 1	1 / 1	3 / 1	3 / 1	– / –	– / –	– / –	– / –
Delete $_{(\neg\text{Contained})}$	1 / 1	1 / 0	1 / 1	1 / 0	1 / 1	1 / 0	1 / 1	1 / 0
Delete $_{(\text{Contained})}$	1 / 1	1 / 1	2 / 1	2 / 1	1 / 1	1 / 1	1 / 1	1 / 1
Total	**8 / 7**	**7 / 4**	**12 / 7**	**11 / 4**	**7 / 6**	**6 / 3**	**7 / 6**	**6 / 3**

Memoizing Element Hash Codes. While Scala's HAMT implementations memoize the full 32-bit hash codes of keys in leaf nodes, neither Bagwell's HAMT design nor Clojure's implementations consider memoization. Adding memoization is trading a higher memory footprint against improved worst case runtime performance of lookup, insertion, and deletion.

Adding Memoization of Element Hash Codes to CHAMP. We strive for a flexible design that can be used with or without memoization. To add memoization to CHAMP, we use a technique called *field consolidation* [GS12]. Instead of storing cached hash codes in a large number of leaf nodes as Scala does, we consolidate the hashes of all elements of a node, to store them in a single integer array per node.

With MEMCHAMP we will refer throughout the text to the variant of CHAMP that adds memoized element hash codes, but drops incremental collection hash codes.

Performance Implications of Memoization

Table 3.1 summarizes the worst case number of `hashCode` and `equals` calls per lookup or update operation. It shows the cross product of all combinations from hash-set and hash-map data types paired with feature options of memoizing and incrementally updating the collection hash code, and memoizing the hash codes of the keys.

As in general with hash-based data structures, each operation needs to calculate the hash code of the key. Non-memoized HAMTs call in worst case `hashCode` once more with Insert $_{\neg\text{Contained}}$: if a prefix collision occurs the `hashCode` of the already stored element must be recalculated to extend the prefix tree. Otherwise, only incremental map operations cause additional `hashCode` calls, because suddenly the hash codes of the values are eagerly evaluated. Furthermore, equality is checked at most once for keys. Memoization avoids some `equals` calls due to fast-failing on equal hashes. Like Clojure and Scala, also other JVM collection libraries differ in their design choices for hash code memoization. E.g., Google's Guava[†] collections do not memoize key hash codes, whereas Java Development Kit (JDK) collections do.

We conclude, based on Table 3.1, that for sets incrementally calculating collection hash codes comes essentially for free: the number of `hashCode` and `equals` calls stay the same. Maps in contrast pay for the eager and incremental evaluation of value hash codes, as compared to lazy evaluation. We suggest considering incremental collection hash codes for maps, if nesting of maps into other hash-based collections is a frequent pattern in a program, library or language.

3.6 Benchmarks: CHAMP versus Clojure's and Scala's HAMTs

We further evaluate the performance characteristics of CHAMP and MEMCHAMP. While the former is most comparable to Clojure's HAMTs, the latter is most comparable to Scala's implementations. Specifically, we compare to Clojure's `PersistentHash{Set,Map}` and Scala's immutable `Hash{Set,Map}` implementations. We used latest stable versions available of Scala (2.11.6) and Clojure (1.6.0) at the time of evaluation.

The evaluated CHAMP data structures are used daily in the runtime environment of the Rascal* programming language, and are currently being tested for inclusion into the Object Storage Model [Wöß+14] of the Truffle language implementation framework. We report on this to assert that our implementations have been well tested and used by users other than ourselves, mitigating threats to the internal validity of the following evaluation.

[†] `https://github.com/google/guava`

* `http://www.rascal-mpl.org`

Assumptions. We evaluate the pure overhead of operations on the data structures, without considering cost functions for `hashCode` and `equals` methods. Our performance assessment is supposed to reveal the overhead of maintaining bitmaps, incremental hash codes and hash code memoization.

Hypotheses. We expect CHAMP's runtime performance of lookup, deletion, and insertion to equal Clojure's and Scala's runtime performance, albeit canonicalizing and computing over two bitmaps. Runtimes should not degrade below a certain threshold —say 20 % for median values and 40 % for maximum values would just be acceptable— (Hypothesis 1).
In general we are betting on significantly better cache behavior and therefore expect to see speedups in iteration (Hypothesis 2). We further expect structural equality checking of two equal collections that do not share reference equal objects to yield at least the same —and likely even higher improvement— as iteration (Hypothesis 3). We also expect that equality checks on derived sets and maps, i.e., measuring `coll.insert(x).delete(x).equals(coll)`, to be orders of magnitude faster (Hypothesis 4).

Despite the addition of a second bitmap, we expect CHAMP maps to use less memory than Clojure's maps because we avoid empty slots (Hypothesis 5). We expect to save a significant amount of memory in big HAMTs because many nodes should have more than two empty slots.

The following hypothesis (Hypothesis 6) formulates our expectations with respect of memory savings of the two HAMT designs: internal values versus leaf nodes. To get a precise expectation we modeled and computed the growth of HAMT-based sets and maps on the Java code skeleton from Figure 3.1. We assume the absence of empty cells in the `contentArray` and consider the JVM's 8-byte memory alignment. Based on our model [SV14], we expect for optimal implementations that HAMTs with internal values require roughly half the memory of HAMTs with explicit leaf nodes, both in 32-bit and 64-bit.

With our final hypothesis (Hypothesis 7) we expect the memory footprints of MEMCHAMP to remain close to Clojure's footprints and still vastly improve over Scala's footprints. With field consolidation MEMCHAMP avoids memory gaps due to object alignment, while minimizing the number of worst case `hashCode` and `equals` methods invocations to the same level that Scala's implementations do.

Benchmark Selection. We assess the performance characteristics of CHAMP with microbenchmarks, focusing on the usage of sets and maps to store and retrieve data and to manipulate them via iterations over the entire collection. We deliberately chose to not use existing CPU benchmark suites or randomly selected applications. The reasons are:

- The pressure on the collection API is quite different per selected real-world application. We have no background theory available to randomly select representative applications which use immutable collections without likely introducing a selection bias.

- It is possible to accurately isolate the use of collections and their operations easily in our microbenchmark setup to confirm or deny our hypotheses. Using these results the designers of standard libraries for programming languages as well as their users will be able to make an informed designed choice for the set and map data structures, irrespective of the other important parts of their designs.

So, on the one hand the importance of these optimizations is different for every application and this remains a threat to external validity. On the other hand the results in the following experiments are very general since they hold for every immutable set or map implementation that uses the proposed optimizations.

Experiment Setup

We use a machine with Linux Fedora 20 (kernel 3.17) and 16 GB RAM. It has an Intel Core i7-2600 CPU, with 3.40 GHz, and an 8 MB Last-Level Cache (LLC) with 64-byte cache lines. Frequency scaling was disabled.

We used Oracle's JVM (JDK 8u25) configured with a fixed heap size of 4 GB. We measure the exact memory footprints of data structures with Google's memory-measurer library.[†] Running times of operations are measured with the Java Microbenchmarking Harness (JMH), a framework to overcome the pitfalls of microbenchmarking.[‡] For all experiments we configured JMH to perform 20 measurement iterations of one second each, after a warmup period of 10 equally long iterations. For each iteration we report the median runtime, and measurement error as Median Absolute Deviation (MAD), a robust statistical measure of variability that is resilient to small numbers of outliers. Furthermore, we configured JMH to run the Garbage Collector (GC) between measurement iterations to reduce a possible confounding effect of the GC on time measurements.

Because each evaluated library comes with its own API, we implemented facades to uniformly access them. In our evaluation we use collections of sizes 2^x for $x \in [1, 23]$. Our selected size range was previously used to measure the performance of HAMTs [Bag01]. For every size, we fill the collections with numbers from a random number generator and measure the resulting memory footprints. Subsequently we perform the following operations and measure their running times:

[†] `https://github.com/DimitrisAndreou/memory-measurer`
[‡] `http://openjdk.java.net/projects/code-tools/jmh/`

Lookup, Insert and Delete: Each operation is measured with a sequence of 8 random parameters to exercise different trie paths. For Lookup and Delete we randomly selected from the elements that were present in the data structures.[§] For Insert we ensured that the random sequence of values was not yet present.

Lookup (Fail), Insert (Fail) and Delete (Fail): Measuring unsuccessful operations. The setup equals the aforementioned setting, however with the difference that we swap the sequences of present/not present parameters.

Iteration (Key): Iterating over the elements of a set or the keys of a map respectively.

Iteration (Entry): Iterating over a map, yielding tuples of type `Map.Entry`.

Equality (Distinct): Comparing two structurally equal data structures. The two object graphs are distinct from each other (i.e., they contain no reference equal elements).

Equality (Derived): Comparing two structurally equal data structures. The second structure is derived from the first by applying two operations: inserting a new element and then removing it again.

We repeat the list of operations for each size with five different trees, starting from different seeds. This counters possible biases introduced by the accidental shape of the tries and it also mitigates a threat to external validity: the shape of a tree depends on the hash codes and hash code transformations which may vary between implementations or applications. E.g., Scala's HAMTs apply a bit-spreading transformation to every hash code —similar to `java.util.HashMap`— to counter badly implemented hash functions.

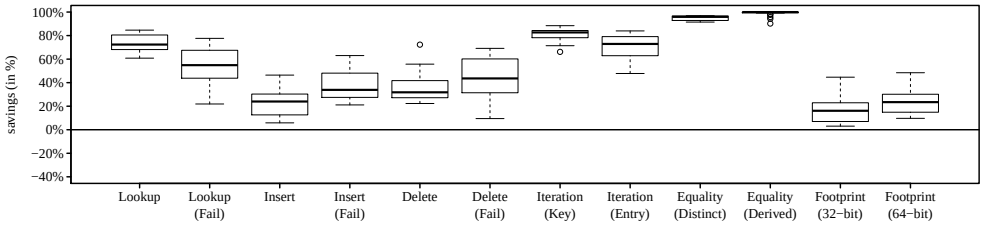
Evaluating HAMT-based sets and maps containing simply random integers accurately simulates any application for which the elements have good uniformly distributed hash codes. A worse-than-uniform distribution would —regardless of the HAMT library— overall reduce the memory overhead per element and increase the cost of updates (both due to clustering of elements). We consider a uniform distribution the most representative choice for our comparison.

We first discuss how CHAMP compares against Clojure and Scala (Sections 3.6 and 3.6), before we focus on the performance implications of adding memoization (Section 3.6).

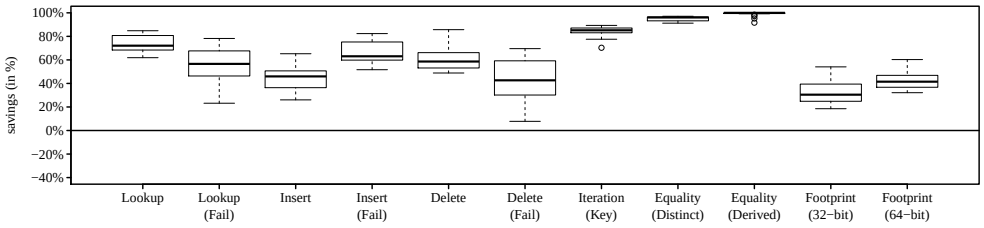
Runtime Speedup Results

We first report the precision of the individual data points. For 99 % of the data points, the relative measurement error amounts to less than 1 % of the microbenchmark runtimes, with an overall range of 0–4.9 % and a median error of 0 %.

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

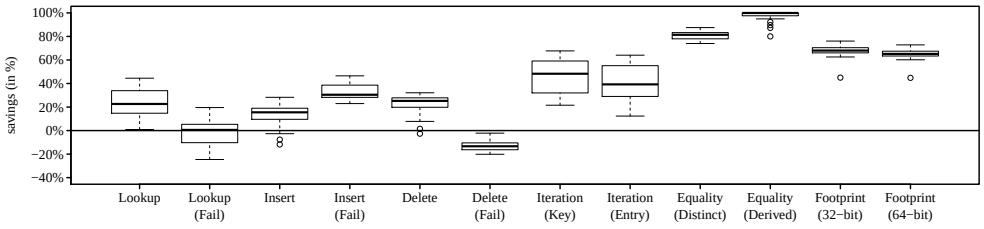


(a) CHAMP versus Clojure's `PersistentHashMap`

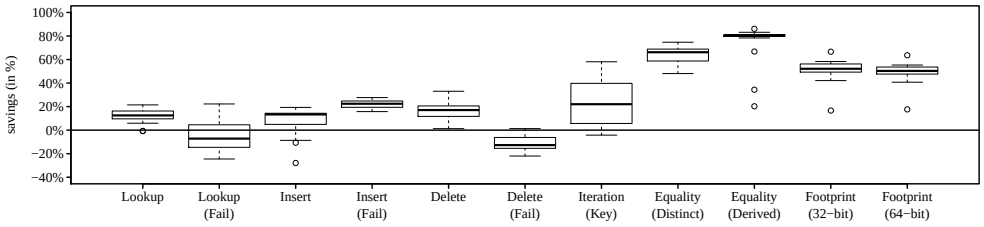


(b) CHAMP versus Clojure's `PersistentHashSet`

Figure 3.6: Runtime and memory savings of CHAMP compared to Clojure's `PersistentHash{Map,Set}`.



(a) CHAMP versus Scala's `immutable.HashMap`



(b) CHAMP versus Scala's `immutable.HashSet`

Figure 3.7: Runtime and memory savings of CHAMP compared to Scala's `immutable.Hash{Map,Set}`.

We summarize the data points of the runs with the five different trees with their medians. Then Figure 3.6(a), 3.6(b), 3.6(a), and 3.6(b) report for each benchmark the ranges of runtime improvements and memory footprint reductions. Each boxplot visualizes the measurements for the whole range of input size parameters. Because the input sizes are scaled exponentially we report savings in percentages using the following formula, normalizing the size factor:

$$(1 - \text{measurement}_{\text{CHAMP}} \,/\, \text{measurement}_{\text{Other}}) * 100$$

Speedups Compared to Clojure's Maps. In every runtime measurement CHAMP is better than Clojure. CHAMP improves by a median 72 % for Lookup, 24 % for Insert, and 32 % for Delete. At iteration and equality checking, CHAMP significantly outperforms Clojure. Iteration (Key) improves by a median 83 %, and Iteration (Entry) by 73 %. Further, CHAMP improves on Equality (Distinct) by a median 96 %, and scores several magnitudes better at Equality (Derived).

Speedups Compared to Clojure's Sets. The speedups of CHAMP for sets are similar to maps across the board, with exception of insertion and deletion where it scores even better.

Speedups Compared to Scala's Maps. Lookup performance improves by a median 23 %, especially at small collections until 128 entries (34–45 %). For bigger sizes the advantage is less pronounced. Given that in practice most collections are small [MS07] and that a usual workload performs more queries than updates, these improvements look promising. For Lookup (Fail) both implementations are neck-and-neck, however CHAMP performs better on the smaller half of the data set and Scala on the bigger half.

Insertion improves by a median 16 %, however with a different pattern from lookup. Insertion performs up to 12 % worse at small collections until 2^5 entries, and then improves again (9–28 %). At Insert (Fail) CHAMP improves across the size range (23–47 %).

Deletion performs with a median runtime reduction of 25 % better than Scala, despite of the compaction overhead. Only for size 2^4 CHAMP is 3 % slower. For Delete (Fail) however, CHAMP is behind Scala all-over (13 % median).

For iteration and equality checking, CHAMP clearly improves upon Scala's immutable hash maps. Iteration (Key) improves by 48 % and Iteration (Entry) by 39 %. Equality (Distinct) improves by a median 81 %. Equality (Derived) unsurprisingly performs 100 % better, because the operation is in a different complexity class.

Speedups Compared to Scala's Sets. The results for sets exhibit similar patterns as the results for maps, however as expected the runtime and memory savings are

across the board are slightly lower than for maps. Median improvements for Lookup (13 %), Insert (13 %), and Delete (17 %) highlight that the performance of those operations are similar to Scala. In contrast, CHAMP performs worse in benchmarks Lookup (Fail) and Delete (Fail). In case of the former, CHAMP lags 7–24 % behind Scala from sizes 2^7 upwards, and up to 22 % across the whole size range in case of the latter.

For all other benchmarks we have a handful of measurements where Scala does slightly better. Lookup performed 1 % worse at sizes 2^{20} and 2^{21}. Insert performed up to 11 % slower until 2^4 elements and 28 % slower at size 2^5.

Iteration (Key) improves by a median 22 % and increases up to 58 % for sets bigger than a thousand elements. However, falsifying our hypothesis, iteration performed 4 % worse at size 2^1 and at three sizes (smaller than 128) 1 % slower. The median improvements of 66 % for Equality (Distinct) and 80 % for Equality (Derived) are less pronounced than for maps, but still substantial. Scala implements set equality with a structural `subsetOf` operation that exploits the HAMT encoding and therefore performs well. Scala's subset operation is conceptually a unidirectional version of structural equality.

Memory Improvements

CHAMP reduces the footprint of Clojure's maps by a median 16 % (32-bit) and 23 % (64-bit), ranging from 3 % to 48 %. The median reductions for sets are 30 % (32-bit) and 41 % (64-bit) respectively, ranging from 19 % to 60 %.

Compared to Scala's maps, CHAMP saves a median 68 % (32-bit) and a median 65 % (64-bit) memory. Savings range from 45 % to 76 %. Note that CHAMP supports a transient representation for efficient batch updates, like Clojure does, and therefore uses one more reference per node. For completeness' sake we also tested a version of CHAMP without support for transients —similar to Scala in terms of functionality— that yields median savings of 71 % (32-bit) and 67 % (64-bit).

CHAMP reduces memory footprints over Scala's sets by a median 52 % (32-bit) and 50 % (64-bit). Once again, a CHAMP variant without support for transients yields slightly higher savings of 57 % (32-bit) and 53 % (64-bit) respectively.

Performance Implications of Adding Memoization

Adding memoization of the full 32-bit hash codes improves worst case performance of lookup, insertion and deletion in practice (cf. Table 3.1). In our microbenchmarks this effect is not observable because we evaluate the performance overhead that incurs with aforementioned operations. Due to measuring overhead, we can evaluate how the performance characteristics of operations change with MEMCHAMP.

Figures 3.8(a), 3.8(b), 3.8(a), and 3.8(b) show the results of comparing MEMCHAMP against Clojure's and Scala's data structures. All-over the memory footprint advantage lessens, due to the additional (consolidated) array of hash codes per node. As a consequence, MEMCHAMP consumes a median 15 % (and maximally 35 %) more memory than Clojure's maps. MEMCHAMP still outperforms Clojure in all operations, with exception of single data points at Insert for maps.

Comparing to Scala, MEMCHAMP retains memory footprint reductions of at least 49 % for maps at sizes bigger than 2^2, and reductions of 15–51 % for equally sized sets. The only outliers here are measurements for size 2^1 where the additional array has a negative impact. The runtimes of lookup and update operations show a similar profile —but worse net runtimes— than CHAMP. MEMCHAMP's allover performance declines, although median runtimes of aforementioned operations are still close to Scala, with exception of Delete (Fail). The reasons for this are twofold. First, locality decreases because hash codes are stored apart from the elements in a separate array. Second, maintaining a second array increases the runtimes of copying operations.

Summary

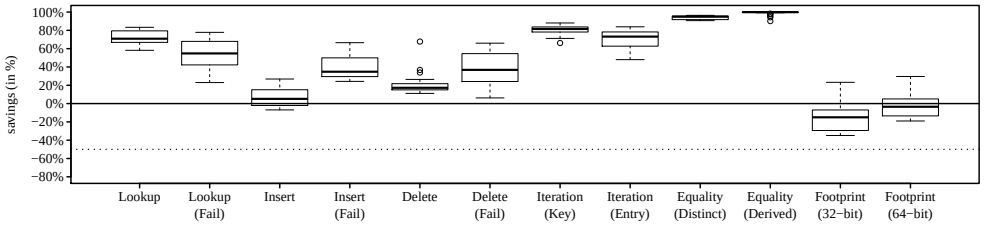
Hypothesis 1 has to be answered case-by-case. With respect to Clojure, it is confirmed. Both variants of CHAMP outperform Clojure's implementations of lookup, insert, and delete. When compared to Scala, the two inherently different designs reveal varying performance profiles. Hypothesis 1 is confirmed for CHAMP, because it performs mostly faster and not often a bit slower than Scala. The exception is when calling delete with a key that is not present in the data structure. Finally, for MEMCHAMP the hypothesis is falsified, because several data points violate our thresholds of acceptable loss of runtime performance. With matching characteristics in terms of `hashCode` and `equals` calls, MEMCHAMP loses runtime performance over Scala, to gain significant memory savings (cf. Hypothesis 7).

Hypothesis 2 is confirmed as well. Over all implementations, the median speedups for iteration range from 22–85 %. However, counter to our expectations CHAMP performs up to 4 % worse on some small sets when compared to Scala.

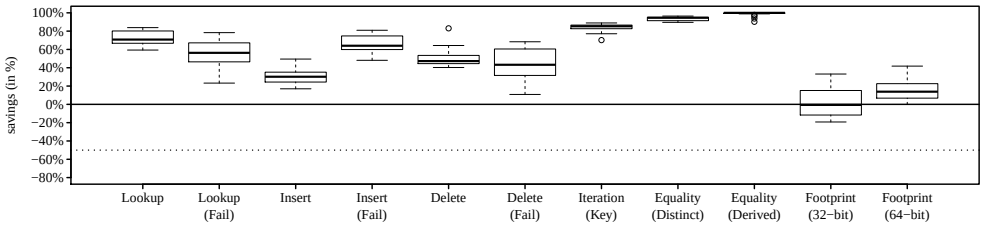
Hypothesis 3 is confirmed. Structural equality checking of two collections that do not share reference equal objects improves by 96 %, 96 %, 81 % and 66 % (medians) over the competing HAMT implementations.

Hypothesis 4 is confirmed. Structural equality checking of two derived collections improves by median 80–100 %, with speedups up to 34 x.

Hypothesis 5 is confirmed. Despite increasing the memory footprint per node, CHAMP-based maps decrease overall memory footprints by up to 46 % compared to Clojure's maps. We conclude that savings due to more efficient compaction outweigh the overhead of the addition of a second bitmap.

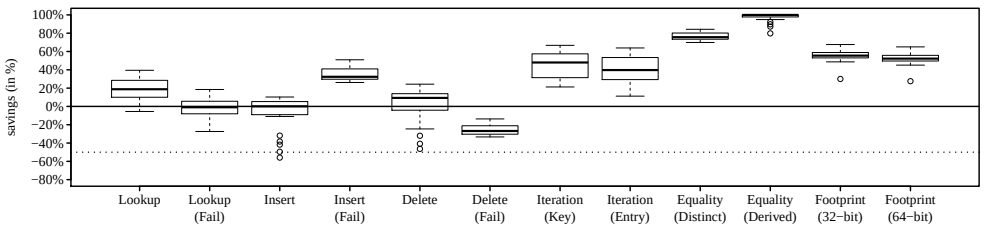


(a) MEMCHAMP versus Clojure's `PersistentHashMap`

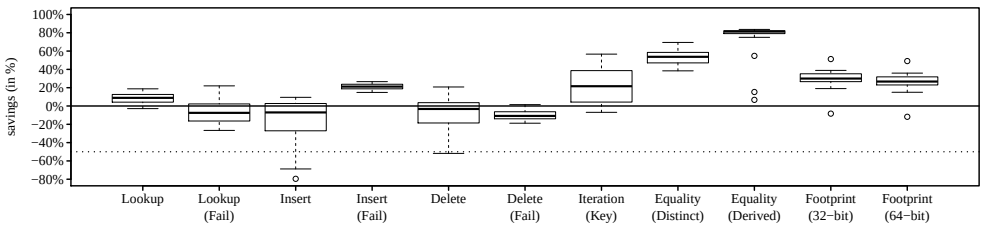


(b) MEMCHAMP versus Clojure's `PersistentHashSet`

Figure 3.8: Runtime and memory savings of MEMCHAMP compared to Clojure's `PersistentHash{Map,Set}`.



(a) MEMCHAMP versus Scala's `immutable.HashMap`



(b) MEMCHAMP versus Scala's `immutable.HashSet`

Figure 3.9: Runtime and memory savings of MEMCHAMP compared to Scala's `immutable.Hash{Map,Set}`.

Hypothesis 6 is confirmed. When compared to Scala's design with leaf nodes, CHAMP reduces memory footprints by median 65 % for maps and 50 % for sets.

Hypothesis 7 is confirmed as well. MEMCHAMP adds little memory overhead over Clojure's implementation, for a great part also due to our savings from Hypothesis 6. The median savings over Scala's HAMTs still range from 27 % to 56 %.

To conclude, despite its more complex encoding, CHAMP achieves excellent runtimes across all tested operations. MEMCHAMP does add overhead over CHAMP, nevertheless runtimes of lookup, insertion, and deletion and memory footprints remain competitive. The significant improvements of the runtime performance of iteration and structural equality checking are observable for CHAMP and MEMCHAMP.

3.7 Case Study: Static Program Analysis

Next to microbenchmarks which isolate important effects experimentally, we also need to evaluate the new design on a realistic case to be able to observe its relevance in relation to other (unexpected) factors. In contrast to the microbenchmarks that exclude a cost model for `hashCode` and `equals` methods, the realistic case has costs attached to those methods. We chose to use a classic algorithm from program analysis which is used in optimizing compilers, static analysis, reverse engineering tools and refactoring tools: computing the control flow dominators [ASU86].

Instead of implementing an optimized data structure specifically for the purpose of computing dominators on a Control-Flow Graph (CFG) [CHK06] we picked a most direct implementation finding a maximal solution to the two dominator equations with Scala's and Clojure's HAMTs, and CHAMP:

$$\mathrm{Dom}(n_0) = \{n_0\}$$
$$\mathrm{Dom}(n) = \left(\bigcap_{p \in \mathrm{preds(n)}} \mathrm{Dom}(p) \right) \cup \{n\}$$

The benchmark source code* uses only classical set and relational calculus in a fixed-point loop: Dom and preds are implemented as maps, the big intersection is generated by first producing a set of sets for the predecessors and then folding intersection over it. The goal is to show that such a direct implementation is viable when implemented on top of an optimized representation of immutable sets and maps. In program analysis pipelines simple high-level code, immutability and persistency are beneficial especially for intermediate data structures like dominator sets. It avoids any bugs caused by unnecessary control and data dependencies.

*`http://michael.steindorfer.name/papers/oopsla15-artifact`

Table 3.2: Runtimes of Clojure, Scala, and CHAMP for CFG dominators experiment per CFG count. All libraries are unmodified.

#CFG	Clojure	Scala	CHAMP	Speedup w.r.t.	
				Clojure	Scala
4096	1686 s	2654 s	170 s	9.9 x	15.6 x
2048	834 s	1387 s	81 s	10.2 x	17.0 x
1024	699 s	1215 s	61 s	11.4 x	19.8 x
512	457 s	469 s	27 s	16.7 x	17.1 x
256	403 s	418 s	18 s	22.3 x	23.1 x
128	390 s	368 s	14 s	28.1 x	26.5 x

For our experiment, we obtained all the ±5000 control flow graphs for all units of code (function, method and script) of Wordpress,[†] one of the most popular open-source Content Management Systems written in PHP, using the PHP AiR framework [HK14]. We then applied the aforementioned dominator implementations to measure CPU time, with JMH, on a randomly-sampled subset of all CFGs. Sample sizes ranged from 128 to 4096 in exponential steps; we omitted smaller samples due to the expected long tail distribution of the CFGs.

This experiment is not trivial since the effect depends on the shape of the real data and the hard-to-predict dynamic progression of the algorithm as it incrementally solves the equations. The nodes in the CFGs we use are the Abstract Syntax Trees of blocks inside PHP units which are arbitrarily complex; equality checks on these sub-structures could overshadow the computations. The hypothesis is that this case should highlight our optimizations showing a significant benefit; if not it will invalidate the relevance of our contributions.

Results

Table 3.2 shows the mean runtimes of 10 benchmark executions. Measurement errors for Clojure and CHAMP are smaller than 1 s, Scala's measurements varied by 8–18 s.

CHAMP's runtimes range from 14–170 s, Clojure ranges from 390–1686 s, and Scala from 368–2654 s. To summarize, CHAMP computed the biggest CFG sample of size 4096 more than two times faster than Clojure and Scala could compute the smallest sample of size 128. Overall, the speedups range from minimal 9.9 x to 28.1 x. The highest speedups were achieved at smaller sample sizes.

[†] `https://wordpress.com`

Table 3.3: Runtimes of Clojure, Scala, and CHAMP for CFG dominators experiment per CFG count. Scala and CHAMP were modified to calculate hash codes lazily, such as Clojure does.

#CFG	Clojure	Scala	CHAMP	Speedup w.r.t.	
				Clojure	Scala
4096	1686 s	1022 s	565 s	3.0 x	1.8 x
2048	834 s	535 s	289 s	2.9 x	1.8 x
1024	699 s	461 s	243 s	2.9 x	1.9 x
512	457 s	277 s	153 s	3.0 x	1.8 x
256	403 s	241 s	132 s	3.1 x	1.8 x
128	390 s	228 s	123 s	3.2 x	1.8 x

Instrumented Libraries to Normalize Results. By profiling and code inspection we identified differences in how the libraries implement the hash code operation. Instead of caching, Scala always recomputes the hash code for collections. This necessarily follows from its design choice where every internal trie node implements the full container API: caching hash codes on every level would incur a major memory overhead. In contrast, Clojure computes collection hash codes lazily, whereas CHAMP incrementally updates them.

These differences do not influence the microbenchmarks since they do not invoke hash code calculations at all. Nevertheless, from the dominators case we conclude that caching hash codes is a critical design element of immutable nested collections. To remove possible bias from hash code calculations, we manually modified the source code of Scala and CHAMP to calculate hash codes lazily such as Clojure does. Subsequently, we ran the dominator experiment again; the results are illustrated in Table 3.3. CHAMP improves over Clojure by median 3 x, and over Scala by 1.8 x. These speedups are directly accountable to improved iteration and equality checking performance in this realistic case.

Conclusion

Although the dominators case was selected to demonstrate a positive effect of our optimizations, it is real and the evaluation could have produced a contra-indication of the relevance of the new CHAMP data structure. It is not true that every algorithm will benefit from CHAMP, but this case does provide a strong indication that if you start from the design decision of using functional programming abstractions, then CHAMP is bound to be faster than the traditional renderings of a HAMT on the JVM.

Table 3.4: Preliminary measurements of Last-Level Cache misses for data structures of size 2^{23}. The number in brackets illustrate how much CHAMP reduces cache misses over the other implementations.

Operation	Last-Level Cache Misses for Maps		
	CHAMP	**Scala**	**Clojure**
Equality (Distinct)	112 682	364 452 (3.2 x)	157 240 (1.4 x)
Equality (Derived)	82 744	351 063 (4.2 x)	146 735 (1.8 x)
Iteration (Key)	110 397	333 985 (3.0 x)	152 210 (1.4 x)
Iteration (Entry)	109 979	341 010 (3.1 x)	147 221 (1.3 x)

Table 3.5: Preliminary measurements of Last-Level Cache misses for data structures of size 2^{23}. The number in brackets illustrate how much CHAMP reduces cache misses over the other implementations.

Operation	Last-Level Cache Misses for Sets		
	CHAMP	**Scala**	**Clojure**
Equality (Distinct)	100 576	205 710 (2.0 x)	203 176 (2.0 x)
Equality (Derived)	71 348	171 872 (2.4 x)	206 862 (2.9 x)
Iteration (Key)	99 268	205 177 (2.1 x)	160 346 (1.6 x)

3.8 Analysis and Threats to Validity

The experimental results in the previous sections answer our hypotheses. From this we learn that the optimizations work but not exactly why they work. The three evaluated HAMT implementations, and their map and set implementations do not only differ from each other, but also from Bagwell's original. In this section we dig deeper to find confirmation of the hypothesis that indeed better locality is the cause of the improvement and we discuss which other factors may be at play to threaten the validity of this claim.

Differences with Clojure's Implementation

Firstly, Clojure uses a lazy sequence abstraction for their iterators. This extra indirection might cause a performance issue or higher number of cache misses as

well. However, we did isolate the effect of our optimizations comparing versions of CHAMP itself (cf. Section 3.8), mitigating this threat to validity.

Secondly, Clojure's `PersistentHashSet` internally wraps a `PersistentHashMap` instead of specializing for the lack of a value reference. This explains why memory savings are bigger for sets than for maps compared to CHAMP, but has no effect on our conclusions otherwise.

Finally, Clojure uses utility functions for calculating hash codes that dispatch (with two `instanceof` checks) on specific interfaces, to apply specialized hash functions. In our case, Clojure delegated to the standard `hashCode` method. However, these utility functions are called in lookup, insert, and delete and may have a small negative effect on performance.

Influence of Internal Values versus Leaf Values

The most fundamental difference between CHAMP and Scala's implementation is how they store the content. Conceptually, Scala's leaf nodes mirror Java's `Map.Entry` instances, and therefore do not require boxing of tuples when iterating over entries. With leaf nodes, Scala's HAMT design stores the content elements exactly one tree level deeper than other HAMTs. Whereas the differences in memory footprints can be directly attributed to this difference in design, the runtime differences for lookup, insertion, and deletion can not. The median differences between CHAMP and Scala (up to 30 %) could be either due to the additional level of memory indirections or because of implementation details.

Modeling Costs of Hash Codes and Equality

The microbenchmarks deliberately focused on the overhead of operations and excluded explicit costs for `hashCode` and `equals` methods. We mitigated this concern by providing a realistic benchmark that has costs attached, and by additionally microbenchmarking MEMCHAMP, a CHAMP variant that exactly matches Scala's memoized design in numbers of `hashCode` and `equals` invocations. Furthermore, if for a particular workload the expected costs for `hashCode` and `equals` are known, one could establish a worst case cost calculation based on Table 3.1 by weighting the operation runtimes.

Isolating the Iteration and Equality Contributions

We internally validated that the speedups for structural equality checking and iteration are due to our contributions by selectively switching on/off code parts. We are using code generators to produce all different variants of CHAMP implementations to mitigate human error.

Thus, we compared our final CHAMP design with a number of variants. E.g., for equality checking we removed the overriding `equals` implementations (cf. Listing 3.5 and Listing 3.6) to fallback to the equality behavior of `java.util.AbstractSet` and `AbstractMap` implementations. These mini-experiments do confirm that the compaction and canonical forms are the main source of performance improvement.

Observing Cache Misses

Most of our hypotheses in the evaluation are based on a claim that CHAMP has better cache behavior, but measuring this on a JVM is not so easy. We used Linux *perf* tool to observe low-level cache statistics.* In particular, we measured the CPU's hardware events for Last-Level Cache (LLC) misses (i.e., how often a data request could not be served any cache) for the experiments from Section 3.6. We used JMH's built-in bridge to *perf* for our cache measurements.

In a preliminary setup we applied *perf* to measure selective data points at a sampling rate of 1000 Hz. Because sampling does not report the exact amounts of LLC misses, we restricted our observations to largest input size of 2^{23} to the following benchmarks: Iteration (Key), Iteration (Entry), Equality (Distinct), and Equality (Derived). We expect from the large input size to see the effects of data locality more accentuated. Tables 3.4 and 3.5 show the results of these experiments. For sets and maps a pronounced effect is observable in terms of cache misses. CHAMP always has fewer cache misses, explaining (at least for a large part) the observed performance benefits. A future more fine-grained analysis uncovering different cache levels may reveal more detail.

Trie Compression and Canonicalization

On insertion Clojure and Scala compress paths, if and only if the full 32-bit hash codes of two or more elements are equal, by directly allocating a hash collision node. Canonicalization, as presented in Section 3.4, does not implement this form of compaction currently.

3.9 Related Work

Trie data structures have been studied since 1959: they were originally invented by Briandais [dlBri59] and named a year later by Fredkin [Fre60]. Bagwell [Bag00] and Olsson and Nilsson [ON07] give an elaborate overview of trie and hash-trie variants and their performance characteristics.

*`https://perf.wiki.kernel.org/index.php/Main_Page`

HAMTs and Persistent Data Structures. In 2001 Bagwell [Bag01] described the HAMT, a space-efficient trie variant that encodes the hash code prefixes of elements, while preserving an upper bound in $\mathcal{O}(log_{32}(n))$ on lookup, insertion, and deletion. Bagwell described a mutable hash-map implementation, but his HAMT design was picked up to implement efficient persistent data structures [Dri+86; Oka99]. Persistency refers to purely functional, immutable data structures that are incrementally constructed by referencing its previous states. The first persistent HAMT implementation can be attributed to Rich Hickey, lead-developer of Clojure.

CHAMP builds directly on the foundations of HAMTs, improving iteration and equality checking runtime performance while still reducing memory footprints.

Functional Lists and Vectors Inspired by HAMTs. After the introduction of HAMTs, Bagwell published about functional list implementations [Bag02] that were evaluated in the context of Common Lisp and OCaml runtimes. A decade later Bagwell and Rompf [BR11] published a techreport about efficient immutable vectors that feature logarithmic runtimes of split and merge operations. Stucki et al. [Stu+15] improved upon the latter and performed a broader scale evaluation.

These immutable vectors are Array Mapped Tries (AMTs) and not HAMTs, because they build prefix trees from the indices of densely filled lists. Nevertheless, the implementation of such vectors take many design cues from HAMTs.

Concurrent HAMTs. Prokopec et al. [Pro+12] 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.

In contrast to Prokopec et al., CHAMP improves a sequential data structure design. However, similar to Clojure's implementation, CHAMP supports edits on a thread-local copy that together with Software Transactional Memory [ST95] can be used to resolve concurrent modifications.

Generative Programming Techniques. In other work, we applied generative programming techniques [Big98; CE00; McI68] to specialize HAMT nodes [SV14]. We discussed how to reduce the number of specializations from a large exponential number to a small subset while still maximizing memory savings. With this techniques we achieved a median decrease of 55 % in memory footprint for maps and 78 % for sets compared to a non-specialized version, but at the cost of 20–40 % runtime overhead of the lookup operation.

Orthogonal to our previous work, Ureche et al. [UTO13] presented a specialization transformation technique called *miniboxing*. Miniboxing adds specializations for primitive JVM data types to Scala, while reducing the generated bytecode.

In contrast to the generative approach, CHAMP achieves memory reductions and speedups without the need for specialization. However, generative programming techniques could be applied to CHAMP to save even more space.

Cache-Aware and Cache-Oblivious Data Structures [LFN02]. Cache-aware data structures store elements in close space proximity if they are supposed to be used together in close time proximity. Cache-aware data structures exploit knowledge about the memory block sizes, cache lines, etc. Thus, research about cache-aware data structures is mostly concerned with (low-level) system programming languages where the engineer has precise control over memory layout. Java does not offer this: everything beside primitive numeric data types are objects that live on the heap.

In contrast, cache-oblivious data structures try to store elements that are accessed closely in time close to each other, without considering details of cache and memory hierarchies. In that sense, CHAMP can be seen as a cache-oblivious HAMT with respect to iteration and equality checking.

Memory Bloat and Improving Mutable Collections. On the side of mutable collections, Gil et al. [GS12] identified sources of memory inefficiencies and proposed memory compaction techniques [GS12] to counter them. They improved the memory efficiency of Java's mutable `Hash{Map,Set}` and `Tree{Map,Set}` data structures by 20–77 % while keeping the runtime characteristics mostly unchanged. In contrast, CHAMP improves runtime performance of immutable collections while also slightly improving memory performance.

3.10 Conclusion

We proposed CHAMP, a new design for Hash-Array Mapped Tries on the JVM which improves locality and makes sure the trees remain in a canonical representation.

The evaluation of the new data structure design shows that it yields smaller memory footprints and that runtimes can be expected to perform at least the same but usually faster on all operations (with the exception of a slight disadvantage at unsuccessful deletes when compared to Scala). We highlight equality checking and iteration in particular which improves by order of magnitude in a number of cases.

CHAMP's design can be used with or without memoization of hash codes and thus reconciles the benefits of Clojure's and Scala's implementations such as small footprints and improved worst case runtime performance. We further showed that memoizing and incrementally updating collection hash codes is a critical design element of nested collections, which can be added to sets without a cost.

The proposed data structures are currently used within the runtime data structures of the Rascal programming language. We expect CHAMP to be relevant for standard libraries of other JVM programming languages as well.

Chapter 4

Specialization for Memory Efficiency

The hash trie data structure is a common part in standard collection libraries of JVM programming languages such as Clojure and Scala. It enables fast immutable implementations of maps, sets, and vectors, but it requires considerably more memory than an equivalent array-based data structure. This hinders the scalability of functional programs and the further adoption of this otherwise attractive style of programming.

In this chapter we present a product family of hash tries. We generate Java source code to specialize them using knowledge of JVM object memory layout. The number of possible specializations is exponential. The optimization challenge is thus to find a minimal set of variants which lead to a maximal loss in memory footprint on any given data. Using a set of experiments we measured the distribution of internal tree node sizes in hash tries. We used the results as a guidance to decide which variants of the family to generate and which variants should be left to the generic implementation.

A preliminary validating experiment on the implementation of sets and maps shows that this technique leads to a median decrease of 55% in memory footprint for maps (and 78% for sets), while still maintaining comparable performance. Our combination of data analysis and code specialization proved to be effective.

This chapter is based on the following published article: Michael J. Steindorfer and Jurgen J. Vinju. 2014. Code Specialization for Memory Efficient Hash Tries (Short Paper). In Proceedings of the 2014 International Conference on Generative Programming: Concepts and Experiences (GPCE 2014). ACM, New York, NY, USA. URL http://dx.doi.org/10.1145/2658761.2658763.

4.1 Introduction

Trie data structures have been studied since more than 55 years, yet major performance improvements in memory usage are still possible using generative programming. Tries are used to implement efficient persistent set and map data structures [Dri+86; Oka99]. They were originally invented by Briandais [dlBri59] and named a year later by Fredkin [Fre60]. Persistency —in this context— means a functional, immutable data structure that is incrementally built by referencing its previous states; the previous state is what is persistent. In 2001 Bagwell [Bag01] described a Hash Array Mapped Trie (HAMT), a space-efficient trie that encodes common hash code prefixes of elements, while preserving an upper bound in $\mathcal{O}(log_{32}(n))$ on lookup, insert, and delete operations. Bagwell's contribution is a corner stone for immutable collection libraries of modern programming languages that run on the Java Virtual Machine, such as Clojure and Scala.

The hash trie design space. Firstly, the current versions of the aforementioned collections libraries can be considered to be quite optimized, yet we need better memory behavior from a HAMT implementation for the sake of scalability. This is one reason why we explored generative programming for specializing the implementation of a HAMT's internal nodes. Secondly there exist a number of very similar uses of HAMT implementation strategies for different kinds of data structures which cannot be modeled using Java generic programming techniques without loss of efficiency.

Hash tries exist in many variations in standard collection libraries of programming languages. These are the variation points:

- **Update semantics:** Hash tries can have *immutable* semantics, *mutable* semantics, or *staged mutability*.
- **Processing semantics:** *sequentially*, *concurrent*, or *in parallel*.
- **Data type semantics:** sets ($element \mapsto boolean$), maps ($key \mapsto value$), and vectors ($index \mapsto element$).
- **Shape of internal nodes:** Hash trie nodes are n-ary (n defaults to 32).

The Scala collection library is split by the following dimensions: *mutable/immutable* and *sequential/parallel/concurrent*. Within these categories there exist distinct data types for *set/map/vector* semantics. The Clojure library contains one implementation for hash trie-based maps and one for hash trie-based vectors, while sets are implemented as wrappers for maps ($key \mapsto boolean$). Wrapping enables reuse, but at the cost of memory efficiency. These choices illustrate the (common) problem with the family: any manual decomposition of one dimension will make variation in the other dimension impractical or even infeasible.

In other words, the hash trie data structure seems like an ideal case for generative programming in the traditional sense [McI68; Big98; CE00]. We expect to both specialize for better efficiency and to factor the common code for ease-of-maintenance. On the other hand, especially the shape of the internal nodes makes the size of the product family very large. Generating the code, loading it, "jitting" it and keeping it in the CPU's caches would all be hard and lead to performance penalty. Instead, we should find a way to limit the number of specializations necessary to achieve better memory behavior without too much losing run-time performance. This is the technical challenge of this work; code generation is an enabler here, but it requires careful design to benefit from it. We contribute the following:

- Statistical analysis of the shape and distribution of hash trie nodes in practice. This evidence guides the selection of product family members to specialize.
- A hash trie specialization layout that reduces the amount of possible specializations from exponential to quadratic. Incorporating results from our analyses, we show how to further restrict the amount of useful specializations.
- Experimental evidence that the above techniques do decrease memory usage of sets by 78% and maps by 55%, while maintaining comparable performance.

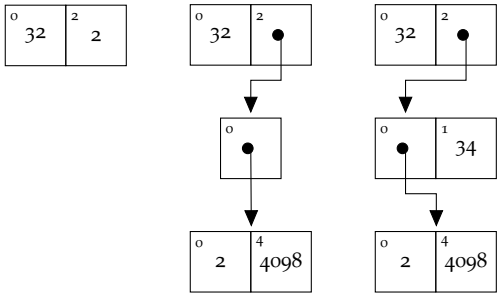
The resulting hash trie family that we can generate appears to significantly outperform the current state-of-the-art implementations in terms of memory consumption.

4.2 Background

A general trie is a lookup structure for finite strings that looks and acts like a finite automaton (DFA) without any loops: the transitions are the elements of the strings, the internal nodes encode prefix sharing, and the accept nodes may point to values associated with the strings. In a hash trie, the strings are the bits of a hash-code of the elements stored in the trie. We use lazily created hash tries, in the sense that internal nodes are only created when two hash-code prefixes would collide, leading to internal nodes with values and references to further "states" stored along.

For example, we sequentially insert objects with the following 32-bit hash-codes into a set: 32, 2, 4098, 34. Figure 4.1 visualizes the hash trie states as these values are inserted. A hash trie distinguishes elements by their hash-code prefixes:

$$\begin{aligned}
32: &\ \ldots 00000\ 00001\ 00000_2 = \ \ldots 0\ 1\ 0_{32} \\
2: &\ \ldots 00000\ 00000\ 00010_2 = \ \ldots 0\ 0\ 2_{32} \\
4098: &\ \ldots 00100\ 00000\ 00010_2 = \ \ldots 4\ 0\ 2_{32} \\
34: &\ \ldots 00000\ 00001\ 00010_2 = \ \ldots 0\ 1\ 2_{32}
\end{aligned}$$



(a) 33 and 2 (b) 4098 (c) 34

Figure 4.1: Inserting numbers into a trie (array indices in top-left).

We have hash tries with a maximal arity of 32 (n-ary trees, with $n = 32$). To select the path sequence that indicates where a value is inserted, we first separate a hash code in chunks of 5 bits (values ranging between 0 and 31).

We expand the tree structure until every prefix can be unambiguously stored. In our example: number 32 is inserted at the root node; number 2 as well (because they do not share a common prefix). Number 4098 shares the prefix path $\rightarrow 2 \rightarrow 0$ with number 2, consequently it is placed unambiguously on level 3. Number 32 shares the prefix $\rightarrow 2$ with numbers 2 and 4098, but can be separated from both on level 2.

Listing 4.1 shows a class skeleton of a hash trie implementation for a set data structure, where the container class, the `TrieSet` contains `size` information and a reference to the `root` node of the trie. The nested `TrieNode` class encodes the possible $n = 32$ sub-tries as a compacted sparse array; the 32-bit integer `bitmap` signals which of the branches are used (for value or child nodes). The size of the array is equal to the number of 1's in the bitmap.

The `contentAndSubTries` array is of type `Object` to either store set elements or references to sub-tries. This compaction technique obviates extra leaf nodes by pulling them up one level. Listing 4.1 closely resembles Clojure's hash trie implementations.

Both the bitmap and the array are candidates for specialization because they introduce overhead. Here, all possible specializations would lead to thousands of classes. Which classes do we generate for maximum effect on footprint with minimal loss on efficiency?

```
abstract class TrieSet implements java.util.Set {
  TrieNode root; int size;
  class TrieNode {
    int bitmap; Object[] contentAndSubTries;
  }
}
```

Listing 4.1: Skeleton of a hash trie-based set data structure in Java.

```
abstract class TrieSet implements java.util.Set {
  TrieNode root;  int size;

  interface TrieNode { ... }
  ...
  class NodeNode extends TrieNode {
    byte pos1; TrieNode nodeAtPos1;
    byte pos2; TrieNode nodeAtPos2;
  }
  class ElementNode extends TrieNode {
    byte pos1; Object keyAtPos1;
    byte pos2; TrieNode nodeAtPos2;
  }
  class NodeElement extends TrieNode {
    byte pos1; TrieNode nodeAtPos1;
    byte pos2; Object keyAtPos2;
  }
  class ElementElement extends TrieNode {
    byte pos1; Object keyAtPos1;
    byte pos2; Object keyAtPos2;
  }
  ...
}
```

Listing 4.2: Skeleton of a specialized hash trie-based set in Java.

4.3 Node Frequency Statistics

We measure the distribution of node arities, to focus specialization on the most frequent node arities. The distribution of arities is governed only by the hash-codes. We can assume a uniform hash distribution, because if the hash-codes in a real application do not approximate a uniform distribution, the trie's efficiency would degenerate anyway due to collisions, and our optimizations would be less relevant.

Table 4.1: Frequencies and cumulative summed frequencies of tree nodes by arity.

Arity	1	2	3	4	5	6	7	8
%	1.44	**63.14**	**14.26**	**3.27**	1.24	0.94	0.93	0.96
∑ %	1.44	64.58	78.84	**82.10**	83.34	84.29	85.21	**86.17**

Arity	9	10	11	12	13	14	15	16
%	1.00	1.05	1.11	1.17	1.23	1.28	1.32	1.33
∑ %	87.17	88.22	89.32	**90.49**	91.72	92.99	94.31	95.65

Arity	17	18	19	20	21	22	23	24
%	1.28	1.09	0.75	0.40	0.15	0.04	0.01	0.00
∑ %	96.93	98.01	98.76	99.16	99.32	99.36	99.37	99.37

Arity	25	26	27	28	29	30	31	32
%	0.00	0.00	0.00	0.00	0.00	0.00	0.01	0.61
∑ %	99.37	99.37	99.37	99.37	99.37	99.38	99.39	100

To generate representative data sets, we use the Java pseudo-random number generator. First, we generated 4096 integers between 0–8M, which we used as target sizes for set data structures.* For each size we then generated a random sequence of integers to be inserted. Note that in Java the hash-code for an integer equals its value. The results in Table 4.1 show how a uniform distribution of hash-codes does lead to a non-uniform distribution in node arities. Similar distributions appear every time we vary the sizes of the sets.

It can be seen that the smaller arities (2–4) account for more than 80% of all nodes. In sum, all node combinations with arities 0–4 account for 82% of all nodes, arities 0–8 for 86%, and arities 0–12 for 90.5%. After this there is a long tail of sizes with a an almost uniform distribution to make up for the last 9.5%.

This data suggests the number of specializations can be reduced to 31, while still achieving an impact on 82% of the nodes. Yet we still need to answer two questions. First, how do the impact numbers translate to real memory savings? Second, should we go beyond arities 0–4 with specializing?

*Bagwell's data set for evaluation included maps up to 8M. For the sake of comparability, we benchmarked the same data points [Bag01].

4.4 Modeling and Measuring Memory Footprints

Measuring JVM memory consumption precisely takes quite some time and energy. To optimize our experiments, instead we accurately model the memory consumption of hash tries, calibrate the model once, and then continue to optimize using the predictive model. We model these two properties:

1. the footprint of trie nodes, following JVM's memory alignment.
2. the overhead of trie nodes compared to real data stored in nodes.

In Java every object is 8-byte memory aligned, allowing for memory compaction techniques [GS12]. Consequently, if an object's header together with the size of all fields do not sum to a multiple of eight, your object will be aligned to the nearest 8-byte boundary and consume more memory than strictly necessary. Note that Oracle's HotSpot JVM uses 12-byte object headers in 32-bit mode and 16-byte headers in 64-bit mode. References consume four bytes in a 32-bit JVM, and eight bytes in a 64-bit JVM. Based on this knowledge we model the footprint (fp) of hash trie nodes in formulas.

$$\text{fp}_{32}(n) = \lceil (12 + 4 + 4)/8 \rceil * 8 + \lceil (12 + 4 + 4 * n)/8 \rceil * 8$$
$$\text{fp}_{64}(n) = \lceil (16 + 4 + 8)/8 \rceil * 8 + \lceil (16 + 4 + 8 * n)/8 \rceil * 8$$

The parameter n is the arity of the node. The first part of each formula calculates the footprint of a tree node which consists of the class header, the size of the integer bitmap and the reference to the array. The second part of the formula describes the layout of an array, containing an integer length field with value n, and n data slots.

We validated the correctness of both formulas on JVM 1.7.0u55, on OS X 10.9.3. To measure the exact footprints of internal trie nodes, we use *memory-measurer*.*

In order to put the numbers obtained from the formulas into perspective, we put them into relation to the theoretical minimum amount of data that has to be stored per node, i.e. the number of references to data/sub-tries. Figure 4.2 shows the overhead per reference a node stores, for each possible arity in 32-bits.† It is visible that the sparse-array implementation (cf. `TreeNode` in Listing 4.1) has a significant overhead at small arities and negligible overhead for the larger arities.

This analysis illustrates how trie nodes usually have low arity with a high overhead. Due to the uniformity of hash-codes, there are low chances of sharing prefixes if there are few elements. When the trie fills up, more and more prefixes are shared, lowering the overhead per element. This is another strong argument in favor of only specializing the classes in the lower ranges.‡

*`https://code.google.com/p/memory-measurer/`

†The picture is comparable for 64-bit mode.

‡*An optimal, uniform hash distribution results in the worst memory performance of hash tries, on the JVM.*

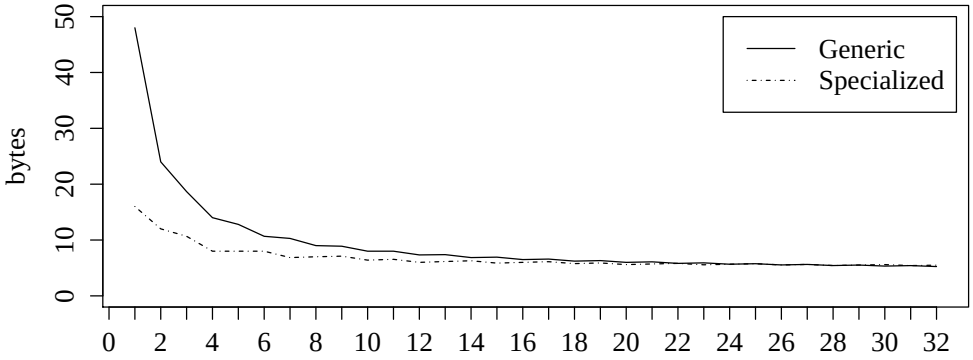


Figure 4.2: Memory overhead per node arity in 32-bit mode.

Our goal is to achieve a worst-case per-reference overhead of 16 bytes in 32-bit mode and 24 bytes in 64-bit mode, even for small numbers of elements. For tries, where nodes are represented as objects, this seems a reasonable target since an object with a single reference to it would consume $12 + 4 = 16$ or $16 + 8 = 24$ bytes in 32-bit and 64-bit mode, respectively.

4.5 Small Number of Specializations for Big Savings

Listing 4.2 contains a Java source code skeleton that shows all specializations for arity 2.* Instead of a single bitmap we now store the 5-bit chunk of the hash-code for each branch in a byte and the reference to either a child or value node in a field.

The following formula models the footprints for these class layouts, yielding our target 24 bytes for size 2 (which was 48 before):

$$\mathrm{fp}^{s}_{32}(n) = \lceil (12 + 1 * n + 4 * n)/8 \rceil * 8$$
$$\mathrm{fp}^{s}_{64}(n) = \lceil (16 + 1 * n + 8 * n)/8 \rceil * 8$$

Given the previous analysis of arity distribution, we save 50% of the memory on at least 80% of the nodes. Figure 4.2 shows how the overhead declines quickly.

The generated code contains specialized implementations of insert and delete methods, such that we scale incrementally to different specialized classes and eventually escalate into the generic implementation. This is the reason that the fields are ordered and we generated, for example, both `ElementNode` and `NodeElement`. The number of specializations necessary for each arity is the number of permutations: 31 for 4, 511 for 8 and 8191 for 12.

*We used Rascal's auto-indenting recursive templates for code generation.

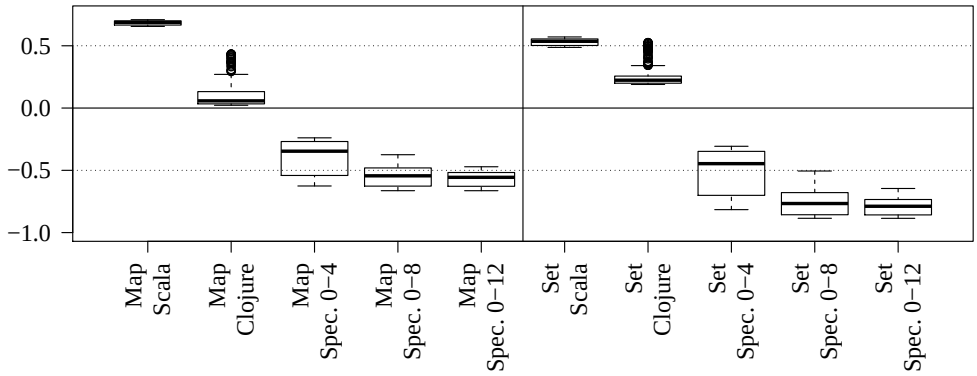


Figure 4.3: Relative footprints of 32-bit sets and maps compared against our generic implementation (i.e., the zero line).

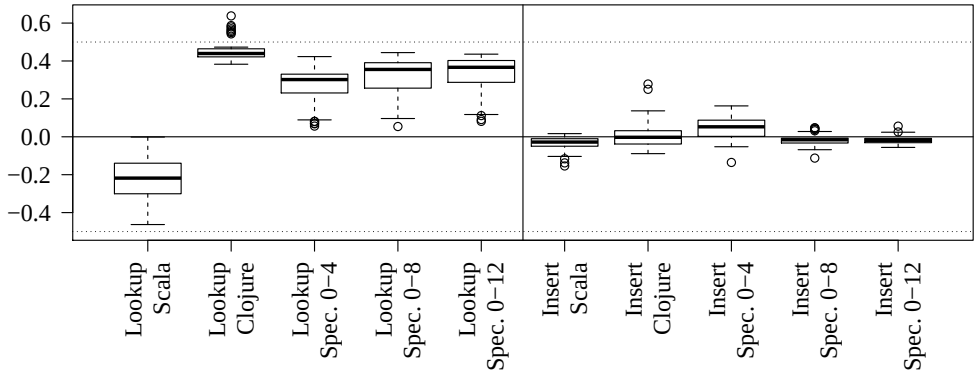


Figure 4.4: Relative run-times for `lookup` and `insert` in maps compared against our generic implementation (i.e., the zero line).

Avoiding Permutations

We know we may save about half of the memory, but at the cost of generating too many classes. A quick experiment showed that this generates too much of an efficiency overhead. Many of the permutation classes we generate have the exact same types and numbers of fields. If we only generate classes for unique combinations of values and internal nodes, then the number of classes would go down to quadratic, or more precisely to $\sum_{i=1}^{n} \binom{2+i-1}{i}$ classes.

We only store how for a certain arity the node splits into content elements and sub-tries. This yields 15 classes (0–4), 45 classes (0–8), and 91 classes (0–12), and all possible specializations would yield 561 classes (0–32). For example, in Listing 4.2 classes `ElementNode` and `NodeElement` would collapse to a single class.

We achieve this complexity reduction by dropping the total ordering of elements within in a specialized node. The cost of this optimization is that we need to dynamically sort the entries when we escalate from a specialized representation to the generic trie node. This only happens for nodes with low numbers of elements, necessarily, so the run-time overhead is expected to be very low, especially given that it only happens at the boundary.

Evaluation

Here we evaluate memory and run-time efficiency. The goal is to position the memory behavior of the resulting tries to the current Clojure and Scala collections, and we report the effect of specialization. The costs of specialization in terms of run-time overhead are also measured.

We used the following versions: *clojure-1.6.0.jar*, *scala-library-2.10.4.jar*, and a research branch of *pdb.values-0.4.1.jar*, our current library. We compared Clojure's `PersistentHash{Set,Map}`, Scala's immutable `Hash{Set,Map}` and our generic set and map against specialized versions with arities up to 4, 8 and 12.

Memory We used the same experimental setup for evaluating the space savings, as we used in Section 4.3 for obtaining frequency statistics. We used the first 256 pseudo-randomly generated target sizes and performed a single experiment for every version. Figure 4.3 shows the results obtained for 32-bit. Our generic trie-map uses less memory than the Scala and Clojure implementations, but the specialized versions still improve on it up to 55% for maps and 78% for sets. The 64-bit measurements are the same, always 5% below the 32-bit results, but exhibit the same magnitude of savings.

Run-time Experiments were run with a 64-bit JVM, version 1.7.0u55, running on an Intel Core i7 3720QM CPU under Mac OS X 10.9.3. We used *Caliper*[†] to run all measurements. We report the median of the 15 last repetitions of each microbenchmark, after Caliper has warmed up the JVM.

For microbenchmarking we reused the earlier setup to generate random sets and maps. For lookup and insert we tested with a sequence of 8 randomly generated examples. The results in Figure 4.4 show how insertion time is not affected much by the postponed sorting and that our implementation performs the same as the

[†] `https://code.google.com/p/caliper/`

Scala and Clojure implementations. For lookup the Scala code is 20% better than our generic implementation, and our specializations cost between 20% and 40% run-time overhead due to the unordered storage of node elements. We still perform faster lookups than the Clojure implementation though.

Specialization range 0–8 yields best performance characteristics while keeping the number of classes necessary low. It performs better than range 0–4 and equally well as range 0–12.

4.6 Conclusion

We reduced the memory footprint of hash trie-based set and map implementations by 55–78% by (one-time) generating specializations of hash trie nodes at the cost of a 40% slow-down in lookup efficiency and no loss in efficiency of insertion.

A side-effect was the generation of a product family for parts of the multi-dimensional design space for hash trie implementations of collections. The presented library is currently used in the run-time of the Rascal meta-programming language.

As future work we intend to study and evaluate a number of additional optimizations to the current design, mainly focusing on run-time efficiency. This would also include a more in-depth evaluation based on wider and more realistic benchmarks.

Chapter 5

A Flexible Encoding for Heterogeneous Data

An immutable multi-map is a many-to-many thread-friendly 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 static analysis of object-oriented systems. When processing such big data sets the memory overhead of the data structure encoding itself is a memory usage bottleneck. Motivated by reuse and type-safety, libraries for Java, Scala and Clojure typically implement immutable multi-maps by nesting sets as the values with the keys of a trie map. Like this, based on our measurements the expected byte overhead for a sparse multi-map per stored entry adds up to around 65B*, which renders it unfeasible to compute with effectively on the JVM.*

We propose a general framework for Hash-Array Mapped Tries which can store type-heterogeneous keys and values: a Heterogeneous Hash-Array Mapped Trie (HHAMT). Among other applications, this allows for a highly efficient multi-map encoding by (a) not reserving space for empty value sets and (b) inlining the values of singleton sets while maintaining a (c) type-safe API.

We detail the necessary encoding and optimizations to mitigate the overhead of storing and retrieving heterogeneous data in a hash-trie on the JVM. Furthermore, we evaluate HHAMT specifically for the application to multi-maps, comparing them to state-of-the-art encodings of multi-maps in Java, Scala and Clojure. We isolate key differences using microbenchmarks and validate the resulting conclusions on a real world case in static analysis. The new encoding brings the per key-value storage overhead down to 30B*: a* 2x *improvement. With additional inlining of primitive values it reaches a* 4x *improvement.*

This chapter is based on an article that is currently under submission: Michael J. Steindorfer and Jurgen J. Vinju. 2016. Fast and Lean Immutable Multi-Maps on the JVM based on Heterogeneous Hash-Array Mapped Tries. arXiv:1608.01036. URL https://arxiv.org/abs/1608.01036.

5.1 Introduction

This chapter is about the challenges of optimizing immutable multi-maps on the Java Virtual Machine (JVM) and how they can be solved using a general method of coding heterogenous hash-array mapped tries. A multi-map is a data structure which 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.

Many-to-many relations or graphs in general occur naturally in application areas such as 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 scale-free networks, like academic citations, the web, online social networks, and program dependence graphs. 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 chapter is that we can exploit these highly common skewed distributions to save memory for the most frequent cases.

On the JVM relations are not natively language-supported; rather the standard libraries of Java, Scala and Clojure either provide implementations of multi-maps, or the map and set Application Program Interfaces (APIs) allow programmers to construct multi-maps easily in a type-safe manner (i.e., using sets as the values of a normal polymorphic map). The goal of this chapter is to overcome the limitations of these existing implementations of multi-maps, improving drastically on the memory footprint without loss of storage, lookup and iteration efficiency. Typically state-of-the-art multi-maps come with a mode of 65B overhead per stored key/value item, while the most compressed new encoding in this chapter reaches an optimum of 30B. In general the encoding has 2 x smaller footprints (modal) when storing reference objects, and 4 x smaller footprints when storing Java primitive values.

On the JVM, immutable collections are used mostly by functional/object-oriented programmers from the Scala and Clojure communities. However, since Java 8 the functional and streaming APIs [Bib+15] are becoming mature, making immutable collections become more relevant in the Java context. Immutability for collections implies referential transparency (without giving up on sharing data) and it satisfies safety requirements for having co-variant sub-types [IV02]. Because of these properties, immutable collections are also safely shared in presence of concurrency.

Our point of departure is the Hash-Array Mapped Trie (HAMT) data structure [Bag01], which has proven to be an efficient immutable alternative to array-based implementations. In contrast to arrays, HAMTs enable fine-grained memory layout optimizations [SV14]. There exists an optimized encoding [SV15] of HAMTs tailored

the JVM, named Compressed Hash-Array Mapped Prefix-tree (CHAMP). The CHAMP data structure allows for time and memory efficient immutable maps and sets. To efficiently encode multi-maps we propose a generalisation of the CHAMP data structure to allow for heterogeneous data shapes. The new resulting data structure, called Heterogeneous Hash-Array Mapped Trie (HHAMT), unifies design elements from both HAMT and CHAMP. A HHAMT allows for a type-safe API in which keys and values can be represented using different types of data within the same map. This allows for all kinds of optimized data structures, but we focus on multi-maps in this chapter as the key purpose. A basic dichotomous HHAMT multi-map is used to either store an inlined single value, or a full nested set data structure. We propose an efficient encoding of HHAMT to mitigate the incurred overhead.

Contributions and Roadmap

We address the design and evaluation of HHAMT as follows:

- Section 5.2 describes the foundations of HHAMT and identifies the main sources of overhead that need to be mitigated.
- Section 5.3 outlines scalable encoding of source code specializations (and their necessary runtime support) to yield memory savings between 2 x and 4 x.
- Section 5.4 compares HHAMT against CHAMP (baseline) to understand the cost of turning a (homogeneous) map into a (heterogeneous) multi-map.
- Section 5.5 compares a specialized HHAMT multi-map against idiomatic solutions from Clojure and Scala.
- Section 5.6 compares the memory footprint of a specialized HHAMT multi-map against state-of-the-art primitive collection libraries (Goldman Sachs, FastUtil, Trove, Mahout).
- Section 5.7 compares the performance of multi-maps in HHAMT, Clojure, and Scala on a realistic case.

Section 5.8 discusses related work and Section 5.9 enumerates further use cases for heterogeneity, before we conclude in Section 5.10. All source code of data structures and the benchmarks discussed in this chapter are available online.*

5.2 From Homogeneity to Heterogeneity

A general trie [dlBri59; Fre60] is a lookup structure for finite strings that acts like a Deterministic Finite Automaton (DFA) without any loops: the transitions are the

*`http://michael.steindorfer.name/drafts/hamt-heterogeneous-artifact`

```
1  abstract class HamtCollection {
2    HamtNode root; int size;
3    // 1-bit + runtime checks (e.g., instanceof)
4    class HamtNode {
5      int bitmap;
6      Object[] contentArray;
7    }
8  }
9  abstract class ChampCollection {
10   ChampNode root; int size;
11
12   // 2-bits (distributed)
13   class ChampNode {
14     int datamap;
15     int nodemap;
16     Object[] contentArray;
17   }
18 }
19 abstract class HeterogeneousHamtCollection {
20    HeterogeneousHamtNode root; int size;
21
22   // n-bits (consecutive)
23   class HeterogeneousHamtNode {
24     BitVector bitmap = new BitVector(n * 32);
25     Object[] contentArray;
26   }
27 }
```

Listing 5.1: Skeletons of a various HAMTs.

characters of the strings, the internal nodes encode prefix sharing, and the accept nodes may point to values associated with the strings. In a HAMT, the strings are the bits of the hash codes of the elements stored in the trie. A HAMT is memory efficient not only because prefixes are shared, but also because child nodes are only allocated if the prefixes of two or more elements overlap.

The first class in Listing 5.1 (lines 1–8) depicts a typical encoding of a HAMT in Java. A single 32-bit integer bitmap is used to encode which of the 32 trie-branches —and correspondingly which slots in the untyped array— are used, together with a mapping function that calculates offsets in the array by counting bits in the bitmap. In general, a HAMT must be able to distinguish between three possible states for each trie-branch: absence of data, and otherwise distinguishing the data category (either payload, or a sub-node). Because a single bit cannot differentiate three different

states, additional dynamic checks —such as **`instanceof`**— are used for discriminating the data category. Note, payload and sub-nodes occur in arbitrary order in the array.

The second class in Listing 5.1 (lines 9–18) depicts the skeleton of the CHAMP encoding [SV15], which operates like a HAMT but uses an explicit encoding to eliminate dynamic **`instanceof`** checks. With two bitmaps CHAMP improves the mapping function to regroup the array slots into two separate homogeneously-typed sequences: a sequence of data payload, followed by a sequence of sub-node references. Because each homogeneous sequence uses its own bitmap, CHAMP kept the bitmap processing identical to HAMTs.

Summary. In a HAMT, each trie node contains an arbitrary mix of data elements and sub-nodes, therefore array slots require type checks individually. In contrast, CHAMP splits HAMT's mixed data sequence into two homogeneous sequences, enabling optimizations that were not possible before. A key to performance —when iterating over or batch-processing elements of homogeneous or heterogeneous data structures— is that individual elements do not need to be checked for its specific type [BDT13]. This is also one of the reasons why the CHAMP performs better than the HAMT. In short: the homogeneous CHAMP data structure provides a good starting point for heterogeneous collections.

Generalizing Towards a Heterogeneous HAMT

The third class in Listing 5.1 (lines 19–27) illustrates the proposed HHAMT skeleton. HHAMT uses a multi-bit encoding like CHAMP but reverts to a sequential representation: one larger bitmap that stores a sequence of 32 n-bit tuples consecutively, instead of at maximum k individual bitmaps. k denotes the maximum number of supported heterogeneous types while n denotes the number of bits needed in our encoding.

For any k, a HHAMT requires $n = \lceil log_2(k+2) \rceil$ bits at minimum per trie-branch to encode all of its possible states. The two additional states are needed for encoding the absence of a trie branch, and encoding sub-trees in case of hash-prefix collisions. For the sake of clarity we mainly focus on the $k = 2$ case in the evaluation (Sections 5.4, 5.5, 5.6 and 5.7), where the required number of bits $n = 2$. This case covers the scenario of distinguishing between a singleton value, and an arbitrarily sized nested set for multi-map implementations. However in the current section we detail the general design and code for arbitrary k. Note that fixing k does influence efficiency trade-offs: experimental results for $k = 2$ do not generalize directly to $k > 2$.

```
interface HeterogeneousMap {
  // pull-based dispatch on type
  <K,V> TypedObject<?> put    (Class<K> keyType, K key, Class<V> valueType, V val);
  <K,V> TypedObject<?> remove (Class<K> keyType, K key);
  <K,V> TypedObject<?> get    (Class<K> keyType, K key);

  // push-based dispatch on type
  <K,V> void put    (Class<K> keyType, K key, Class<V> valueType, V val,
    CallbackMap callbacks);

  <K,V> void remove (Class<K> keyType, K key, Class<V> valueType, V val,
    CallbackMap callbacks);

  <K,V> void get    (Class<K> keyType, K key, Class<V> valueType, V val,
    CallbackMap callbacks);
}

interface TypedObject<T> {
  Class<T> getType();
  T get();
}

interface CallbackMap {
  <E> Consumer<E> put (Class<E> elementType, Consumer<E> consumer);
  <E> Consumer<E> get (Class<E> elementType);
}
```

Listing 5.2: Generic HHAMT interface, based on *Item 29: Consider typesafe heterogeneous containers* of *Effective Java* [Bloo8].

HHAMT API

Although this is not a core contribution, since we model data structures beyond the power of Java's type system, we should detail how to circumvent it. Java does not support union types, and a polymorphic wrapper (such as Scala's `Either`) would introduce overhead. To solve this we can either write or generate specialized code for fixed combinations of types, or use Java's method type *polymorphism* and judicious use of class literals (a.k.a. type tokens like `Integer.class`).

For multi-maps, which are heterogeneous internally, a generic API will suffice. For other applications, such as when the keys or values of a map are type heterogeneous or primitive values are inlined, code generation for the wrapping API is possible.

```
1  public void heterogeneousInterfaceTest() {
2    put(String.class, "abc", int.class, 5);      // accepted by guard condition
3    put(String.class, "abc", Integer.class, 5);  // accepted by guard condition
4
5    put(String.class, "abc", long.class, 5L);    // rejected by guard condition
6    put(String.class, "abc", Long.class, 5L);    // rejected by guard condition
7  }
8
9  static <T, U> void put(Class<T> keyType, T keyInstance, Class<U> valueType, U
       valueInstance) {
10   switch(keyType.getName()) {
11     case "java.lang.String":
12       switch(valueType.getName()) {
13         case "int":
14           put((String) keyType.cast(keyInstance), (int) valueInstance);
15           return;
16         case "java.lang.Integer":
17           put((String) keyType.cast(keyInstance), (Integer) valueInstance);
18           return;
19       }
20   }
21
22   System.out.println("Unsupported Type");
23 }
24
25 static void put(String keyInstance, Integer valueInstance) {
26   System.out.println("put(String keyInstance, Integer valueInstance)");
27 }
28
29 static void put(String keyInstance, int valueInstance) {
30   System.out.println("put(String keyInstance, int valueInstance)");
31 }
```

Listing 5.3: The method `heterogeneousInterfaceTest` illustrates a possible way to map a generalized HHAMT interface to specialized functions with type guards (cf. **`switch`** statement).

If we use Java's method polymorphism (cf. *Effective Java*, Item 29 [Bloo8]) instead we may avoid code generation at a certain cost. We use type tokens and their `cast` method to encode type heterogeneity. Up to Java 8 it is not possible to bind primitive types to type variables though, and care must be taken to avoid dynamic type errors. Casts can be avoided using either-typed (temporary) wrappers or a typed callback interface. Examples are contained in Listings 5.2 and 5.3. Note that the internals of the HHAMT can decide upon the type a value with 100 % certainty.

Bitmap Encoding and Indexing

The heterogeneous skeleton in Listing 5.1 (lines 19–27) does not exhibit an optimal encoding. We specialize the `BitVector` code for obtaining better memory performance. Assuming $k = 2$, we use a single **`long`** field as bitmap, for a larger k we would use several consecutive **`int`** or **`long`** fields.

The way we index into the trie node array (for lookup, insertion or deletion) is a key design element. This indexing is different between the original CHAMP encoding and the new HHAMT encoding because there are k-cases to distinguish.

Listing 5.4 shows how CHAMP's original per-node-bitmap indexing would work if generalized to multiple entry types. By default CHAMP already distinguishes between payload data and nested nodes with separate bitmaps. This baseline (naive) design for heterogeneous hash tries carries on similarly to distinguish more types of references. The masking function (lines 1–3) selects the prefix bits based on the node level in the tree (`shift = 5 * level`). The index function (line 4–6) requires a `bitpos` variable with a single non-zero bit, designating one of the 32 possible branches. It then maps from the `bitmap/bitpos` tuple to a sparse-array index by counting the non-zero bits in `bitmap` on the right of `bitpos`. On line 9 a method template for lookup, insertion, and deletion is shown. Because for each of the three data categories a separate bitmap is used the processing happens in a linear-scanning manner until the right category for a hash-prefix is matched, or the default case applies (line 31).

Although lines 12, 18, and 24 suggest the use of separate bitmaps for each distinct type, two bitmaps are sufficient to distinguish between three cases:

```
int xxxxMap = rawMap1 & rawMap2;
int dataMap = rawMap2 ^ xxxxMap;
int nodeMap = rawMap1 ^ xxxxMap;
```

The above listing depicts how to retrofit three logical bitmaps onto two physical bitmaps. The fields for `datamap` and `nodemap` are renamed to `rawMap1` and `rawMap2`. Subsequently, the data structure infers three logical views from the two raw bitmaps. We further will refer to this retrofitted heterogeneous variant as Heterogeneous Compressed Hash-Array Mapped Prefix-tree (HCHAMP).

Listing 5.5 illustrates operations on the bitmap in the generalized data structure that is specialized to $k = 2$. The mask function can be reused, and the `index` function is scaled to using a **`long`**. The new template method retrieves the 2-bit wide `pattern` (line 12) and translates it to an enum value to switch on. Instead of having to search linearly, as in Listing 5.4, we now jump directly to the relevant case handler. Using a fast **`switch`** is even more beneficial with an increasing number of heterogeneous types ($k > 2$), and while iterating which is when type dispatch will be hot.

```
static final int mask(int hash, int shift) {
  return (hash >>> shift) & 0b11111;
}
static final int index(int bitmap, int bitpos) {
  return Integer.bitCount(bitmap & (bitpos - 1));
}

// processing in (Heterogeneous) CHAMP
void processAtNode(int keyHash, int shift) {
  int mask = mask(keyHash, shift);
  int bitpos = bitpos(mask);

  int nodeMap = nodeMap();
  if ((nodeMap & bitpos) != 0) {
    // process node
    int index = index(nodeMap, bitpos);
    ...code for lookup, insert or delete ...
  } else {
    int dataMap = dataMap();
    if ((dataMap & bitpos) != 0) {
      // process payload category 1
      int index = index(dataMap, bitpos);
      ...code for lookup, insert or delete ...
    } else {
      int xxxxMap = xxxxMap();
      if ((xxxxMap & bitpos) != 0) {
        // process payload category X
        int index = index(xxxxMap, bitpos);
        ...code for lookup, insert or delete ...
      } else {
        // process empty slot
        ...code for lookup, insert or delete ...
    }
  }
}
```

Listing 5.4: Processing of multiple bitmaps with 1-bit entries.

```
static final int index(long bitmap, long bitpos) {
  return Long.bitCount(bitmap & (bitpos - 1));
}

// processing in a Heterogeneous HAMT
void processAtNode(int keyHash, int shift) {
  long bitmap = bitmap();

  int mask = mask(keyHash, shift) << 1;
  long bitpos = 1L << mask;

  int pattern = (int) ((bitmap >>> mask) & 0b11);
  Type type = toEnum(pattern);

  switch (type) {
  case EMPTY:
    ...code for lookup, insert or delete ...
    break;
  case NODE:
    int index = index(filter(bitmap, type), bitpos);
    ...code for lookup, insert or delete ...
    break;
  case PAYLOAD_CATEGORY_1:
    int index = index(filter(bitmap, type), bitpos);
    ...code for lookup, insert or delete ...
    break;
  case PAYLOAD_CATEGORY_2:
    int index = index(filter(bitmap, type), bitpos);
    ...code for lookup, insert or delete ...
    break;
  }
}
```

Listing 5.5: Processing of one bitmaps with 2-bit entries.

```
static final long filter(long bitmap, Type type) {
  long mask = 0x5555555555555555L;

  long masked0 = mask & bitmap;
  long masked1 = mask & (bitmap >> 1);

  switch (type) {
  case EMPTY:
    return (masked0 ^ mask) & (masked1 ^ mask);
  case NODE:
    return masked0 & (masked1 ^ mask);
  case PAYLOAD_CATEGORY_1:
    return masked1 & (masked0 ^ mask);
  case PAYLOAD_CATEGORY_2:
    return masked0 & masked1;
  }
}
```

Listing 5.6: Filtering of multi-bit patterns (for $k = 2$).

Optimizing Bit-Counting

Extra bitwise operations are in the overhead of HHAMT which we need to mitigate. We explain three techniques to do so.

Relative Indexing into a Single Data Category. The purpose of the `index` function in Listing 5.5 is to calculate the relative index of a data element within its data category. Given a `type` enum and a trie-branch descriptor (`bitpos`), the `index` function calculates how often the given `type` pattern occurs in the bitmap before the `bitpos` position.

The Java standard library contains bit count operations for the types **`int`** and **`long`** that count the number of bits set to 1. These functions do not support n-bit patterns with $n > 1$. However, we want to reuse the aforementioned functions, because on the widespread *X86/X86_64* architectures they map directly to hardware instructions. We introduce some bitmap pre-processing with filters to get to that point where we can use the native bit counters. Listing 5.6 illustrates how such a filter reduces a matching 2-bit wide pattern to a single bit set to 1, while resetting all other bits to 0.

Distribution of Heterogeneous Elements. While lookup, insertion, and deletion only require indexing into a single data category, on the other hand iteration and streaming require information about the types of all elements in a trie node: their frequency per node. Studies on homogeneous data structures [BDT13] have shown avoiding checks on a per elements basis is indeed relevant for performance.

To also avoid such checks in HHAMT we introduce the use of histograms, on a per node basis, that are calculated in constant time (for a given branch factor). The computation is independent of the number of heterogenous types:

```
int[] histogram = new int[2^n];

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

The former listing abstracts over the number of heterogeneous elements and executes in 32 iterations. `n` and `mask` are constants, where `mask` has the lowest n bits set to 1. In its generic form, the code profits from compiler-level optimizations, such as scalar replacement [Sta14] to avoid allocating the array on the heap, and loop unrolling.

We assigned the bit-pattern `EMPTY = 0` and `NODE` $= 2^{n-1}$, the various remaining heterogenous types are assigned consecutively in the middle. For iteration, streaming, or batch-processing data, histograms avoid expensive repetition of indexing individual categories: k bit-count operations, where each one requires applying a filter to the bitmap. For example, the total number of elements, regardless of their types, can be calculated with `32 - histogram[EMPTY] - histogram[NODE]`. The otherwise complex code for trie-node iteration reduces to looping through the two-dimensional histogram using two integer indices. The 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 [SV15]. Finally, iteration can exit early when the running counter reaches the maximum branching factor of 32 to avoid iterating over empty positions in the tail. Note that for fixed k the code can be partially evaluated (i.e., by a code generator) to avoid the intermediate histogram completely.

Reversing the Bitmap Encoding: Extracting Index and Type. For enabling fast structural equality comparisons [SV15] maintaining a canonical form of the hash-trie is essential, also after the delete operation. For HHAMT and especially for HHAMT multi-maps this takes an extra effort: the deletion operation does know the index and type of the removed element, however it does not know the index and type of the remaining elements. Upon deletion, canonicalization triggers inlining of sub-tree structures with only a single remaining payload tuple. Efficiently recovering the index an type of the only remaining tuple is important for the overall efficiency of the deletion operation. We devised a recovery function for bitmaps with n-bit tuples, based on Java's standard library functions: `Long.numberOfTrailingZeros(bitmap)/n*n`. By first counting the number of trailing zeros, we approximate the region within the bitmap that contains bit-pattern information. We subsequently adjust the non-zero count to our n-bit pattern alignment with an integer division followed by an

multiplication. As a result, we recovered the `mask` that allows retrieving the type of the remaining element (cf. Listing 5.5, lines 10–13).

Outlook. We have now discussed all techniques to mitigate Central Processing Unit (CPU) overhead caused by a more complex indexing. The remaining challenge is saving memory, which is discussed next.

5.3 Lean Specializations

Specialization for a fixed number of heterogeneous types will prove essential for both memory efficiency and CPU performance. In this section we take the perspective of the general k-heterogeneous HHAMT. The effect of these techniques will be evaluated in Sections 5.4, 5.5, 5.6 and 5.7 in different contexts.

For a HHAMT with k different types, there exist $arity_{nodes} \times \prod_{i=1}^{k} arity_i$ possible strongly-typed variants in theory, with the constraint that $arity_{nodes} + \sum_{i=1}^{k} arity_i <= 32$. We can reduce this complexity by grouping different heterogeneous types together into a section that is represented by their least upper bound type. Ultimately, we can group together all reference types and sub-nodes into one section, and all primitive types into another section [UTO13], to achieve a quadratic upper bound that overcomes the dichotomy of reference and primitive types. Therefore, in the remainder of this section we will focus on the most common case of $k = 2$ that also satisfies our use case of multi-maps. Note that due to the bitmap encoding we always know the precise type of an object, using more general types for internal storage is solely used to reduce the total number of specializations.

There exist empirical evidence [SV14] for $k = 1$ that specializing up to arities of 8 or 12 balances impact on memory performance best with the necessary amount of generated code. However with heterogeneity $k > 1$ this may not hold, and to exploit inlining primitive types for saving more memory we should support specializing the full bandwidth up to 32.

We now present an improved approach for code generation that allows fully specialized collections (i.e., "array-less" data structures) with very low memory footprints. It aims to overcome the following issues that typically compromise performance of specialized code:

Additional Polymorphism: Turning a generic data type into a set of distinct specializations compromises trace-based inlining strategies of a Just-in-time (JIT) compiler. By introducing specializations, previous monomorphic call-sites are turned into polymorphic call-sites: a JIT compiler has to fallback to dynamic dispatch for method method calls that were previously resolved to direct calls.

```
abstract class Set1 implements Set {
  final Object slot0;

  Set add(Object value) {
    if (slot0.equals(value)) {
      return this;
    } else {
      return new Set2(slot0, value);
    }
  }
}
```

Listing 5.7: Interlinking of specializations prohibits generic methods: `Set1` contains a static reference to `Set2`.

Code Bloat: Substituting a dynamic structure with specialiations often demands the specialization of operations as well. In the case of hash-tries, we specialize for constant array sizes [SV14]: instead of referencing a heap structure, we inline the array-fields into a trie-node. Unfortunately the resulting memory saving come at a price: suddenly array operations (i.e., allocation, copy, get, set, length) must be specialized as well.

Interdependency of Specializations: In general, each specialized data type contains static references to other specializations that represent possible next states. Listing 5.7 exemplary lists the `add` method of set data structure specialized for one element that might return a set specialized for two elements. The switching between specialized representations, puts strain on the JIT compiler at run-time due to incremental class loading and the constant need to compile methods of specializations during a data structure builds up, further, it is one source of code bloat.

In the remainder of this section we detail our approach of specialization that remedies the aforementioned overheads. In our design, a specialization represents purely a heterogeneous trie-node, specialized for a certain content size. It contains pre-evaluated content stored in static fields and instance fields (storing the bitmap, and the inlined array content), however does not override methods.

Indexing and Selecting Specializations

We replace the use of arrays, which can be allocated using an arbitrary length parameter, with fields inlined in specialized classes. Commonly, for each specilization a unique constructor must be called (cf. Listing 5.7, specialization interlinking).

Which constructor must be called depends on the current state of a trie node and the operation applied to the data structure.

To enable class selection at run-time, we introduce a global and static two-dimensional `Class[][] specializations` array, indexed by the number of primitive data fields (t) and the number of reference fields (u). This lookup table solves the interdependency problem of specialization: when adding a key-value tuple of reference type the next specialization can be be determined with `specializations[t][u + 2]`, or respectively with `specializations[t - 2][u]` when a tuple of primitive type is deleted. Once a specialization is selected, it can be initialized by invoking its default constructor: `Object instance = specialization[t][u].newInstance()`.

Since the array is often used and relatively small, we found it runs faster than distributing code over the specialized classes. This also allows for more generic code in base classes that is more likely to be optimized by the JIT compiler.

Initializing Instances of Specialized Classes

For the generic representation that operates on arrays, we would use `System.arraycopy` initializing a new trie node, which is really fast. Now we want to try and approach similar efficiency for initializing the fields of our specialized classes.

Our solution is to introduce a `arraycopy`-like operation that is capable of copying consecutive fields between object instances: an `ArrayView` on an object layout is an abstraction which logically maps an arbitrary region within objects to an array. To ensure safety we check whether the JVM indeed maps the fields in a consecutive region at class loading time. Using a primitive `ArrayView.copy` we achieve similar performance to `System.arraycopy`. We measured the effect using a micro-experiment: the new primitive is about 20–30 % faster than field-by-field copying. Since eventually copying trie nodes is the primary bottleneck we may expect similar speedups for insertion- and deletion-intensive code of HHAMT and less for lookup intensive code.

Listing 5.8 shows how we can model an array view of a range of fields within a heap object. Once we have obtained a reference to an `ArrayView`, we can invoke corresponding `(getFrom|setIn)HeapRegionArrayView` methods that either retrieve or a set a value of a `ArrayView`. To mimic `System.arraycopy` on an `ArrayView`, we use `sun.misc.Unsafe.copyMemory`. For our experiments, we extended the `copyMemory` function to support copying from/to objects while catering for card marking, i.e., signaling the Garbage Collector (GC) that references changed.

Relationship to `VarHandle` API of the Upcoming JDK 9. The Java Development Kit (JDK) 9 will introduce an API for uniformly referencing, accessing and modifying fields. Thus, independently of the realization of a variable —static field, instance field, or array— a handle to the field reference can be obtained. In earlier versions of Java, the granularity of references was restricted to objects; a `VarHandle` in contrast

```
class TrieNode_T4_U3 implements Node {
  long bitmap;

  int key0; int val0;
  int key1; int val1;

  Object slot0;
  Object slot1;
  Object slot2;

  static ArrayView getArrayView_T4() {
    return createHeapRegionArrayView(
          TrieNode_T4_U3.class, "key0", "val1");
  }

  static ArrayView getArrayView_U3() {
    return createHeapRegionArrayView(
          TrieNode_T4_U3.class, "slot0", "slot2");
  }
}
```

Listing 5.8: `ArrayView` on regions of specialized trie node.

enables addressing fields or arrays (at a finer granularity) inside an object. The `VarHandle` API furthermore contains abstractions to view and process off-heap memory regions as arrays. However, it does not provide likewise abstractions for obtaining array views on on-heap regions.

The aforementioned `ArrayView` implementation we used provides a proof-of-concept implementation on how to extend the `VarHandle` API to support array views for on-heap regions.

Summary

For collections, we eliminated issues that typically compromise the performance of specialized code. We will evaluate the effects of these techniques in Section 5.4.

5.4 Benchmarks: Heterogeneous Multi-Maps versus Maps

In this section we evaluate the performance characteristics of the various implementations of multi-maps on top of the HCHAMP, HHAMT, and specialized HHAMT encodings, comparing them against the basic homogeneous CHAMP map data structure as state-of-the-art baseline [SV15]. We are interested in isolating the effects that are incurred by adding the heterogeneity feature:

- HCHAMP is close to CHAMP (same logic, but derives three bitmap views from two physical bitmaps);
- HHAMT generalizes the heterogeneous bitmap encoding;
- and specialized HHAMT improves memory footprints by dynamically selecting statically known specializations.

In the case of multi-maps, heterogeneity lies in the internal distinction between $1:1$ and $1:n$ mappings.

Assumptions. We evaluate the pure overhead of operations on the data structures, without considering cost functions for `hashCode` and `equals` methods. This performance assessment should reveal the overhead of adding heterogeneity to CHAMP, the effect of the specialization approach and the effect of accessing the heterogeneous data elements.

Hypotheses. We expect HHAMT's runtime performance of lookup, deletion, and insertion to be similar comparable to CHAMP's runtime performance, but never better. 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 1).

Iteration over a multi-map is more complex than iterating over a map. Iteration (Key) has to distinguish between heterogeneous categories, whereas Iteration (Key) has to distinguish heterogeneous categories, Iteration (Entry) additionally has to flatten nested sets to obtain a tuple view on multi-maps. Consequently, we assume costs of about 25 % for median values and 50 % for maximum values as well (Hypothesis 2).

Based on related work in the domain of specializing HAMTs [SV14], we expect that specializing may introduce run-time overhead. However, we expect lower overhead (than the reported 20–40 % degradations for lookup) due to our mitigation strategies outlined in Section 5.3 (Hypothesis 3).

Furthermore, memory footprints of HCHAMP and HHAMT should in practice match CHAMP's footprints, because all variants use in total 64-bits for bitmaps (Hypothesis 4).

Experiment Setup

We use a machine with Apple OS X (10.11.3) and 16 GB RAM. It has an Intel Core i7-3720QM CPU, with 2.60 GHz, and an 6 MB Last-Level Cache (LLC). Frequency scaling was disabled. For testing, we used an OpenJDK (JDK 8u65) JVM configured with a fixed heap size of 8 GB. We measure the exact memory footprints of data

structures with Google's memory-measurer library.* Running times of operations are measured with the Java Microbenchmarking Harness (JMH), a framework to overcome the pitfalls of microbenchmarking.† For all experiments we configured JMH to perform 20 measurement iterations of one second each, after a warmup period of 10 equally long iterations. For each iteration we report the median runtime, and measurement error as Median Absolute Deviation (MAD), a robust statistical measure of variability that is resilient to small numbers of outliers. Furthermore, we configured JMH to run the GC between measurement iterations to reduce a possible confounding effect of the GC on time measurements.

In our evaluation we use collections of sizes 2^x for $x \in [1, 23]$. Our selected size range was previously used to measure the performance of HAMTs [Bag01; SV15]. For every size, we fill the collections with numbers from a random number generator and measure the resulting memory footprints. Subsequently we perform the following operations and measure their running times:

Lookup, Insert and Delete: Each operation is measured with a sequence of 8 random parameters to exercise different trie paths. For Lookup and Delete we randomly selected from the elements that were present in the data structures.‡ For Insert we ensured that the random sequence of values was not yet present.

Lookup (Fail), Insert (Fail) and Delete (Fail): Measuring unsuccessful operations. The setup equals the aforementioned setting, however with the difference that we swap the sequences of present/not present parameters.

Iteration (Key): Iterating over the elements of a set or the keys of a map respectively.

Iteration (Entry): Iterating over a multi-map, flattening and yielding tuples of type `Map.Entry`.

We repeat the list of operations for each size with five different trees, starting from different seeds. This counters possible biases introduced by the accidental shape of the tries, and accidental bad locations in main memory. Evaluating HAMT data structures containing simply random integers accurately simulates any application for which the elements have good uniformly distributed hash codes. A worse-than-uniform distribution would —regardless of the HAMT-like implementation— overall reduce the memory overhead per element and increase the cost of updates (both due to clustering of elements). We consider a uniform distribution the most representative choice for our comparison.

*`https://github.com/DimitrisAndreou/memory-measurer`

†`http://openjdk.java.net/projects/code-tools/jmh/`

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

Experiment Results

We first report the precision of the individual data points. For 99 % of the data points, the relative measurement error amounts to less than 1 % of the microbenchmark runtimes, with an overall range of 0–4.8 % and a median error of 0 %.

We summarize the data points of the runs with the five different trees with their medians. Then Figure 5.1(a), and 5.1(b) report for each benchmark the ranges of runtime improvements or degradations. For brevity, the effects on memory footprints and of specialization are not contained in the boxplots, but are discussed in text. Each boxplot visualizes the measurements for the whole range of input size parameters. For improvements we report speedup factors above the neutral line ($\text{measurement}_{\text{CHAMP}}$ / $\text{measurement}_{\text{HHAMT-Variant}}$), and degradations as slowdown factors below the neutral line , i.e., the inverse of the speedup equation. From this data we learn the following:

Confirmation of Hypothesis 1: The cost of converting a map to a multi-map stayed within the specified bounds for both HCHAMP and HHAMT.

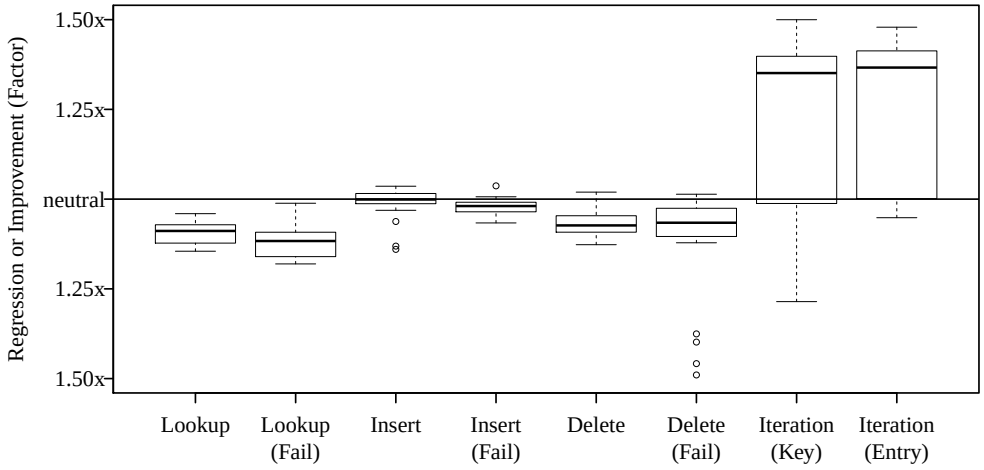
For HCHAMP, Lookup, Insert and Delete added a median slowdown of 9 %, 0 %, and 7 % respectively, and Lookup (Fail), Insert (Fail) and Delete (Fail) added 12 %, 2 % and 7 % respectively. With exception to single outliers produced Delete (Fail), the maximum slowdown are lower than 18 % at most.

For the generalized HHAMT, the costs for multi-maps over maps are higher. Lookup, Insert and Delete added a median slowdown of 20 %, 5 %, and 10 % respectively, and Lookup (Fail), Insert (Fail) and Delete (Fail) added 22 %, 0 % and 13 % respectively. With exception to single outliers produced Delete (Fail), the maximum slowdown are lower than 29 % at most.

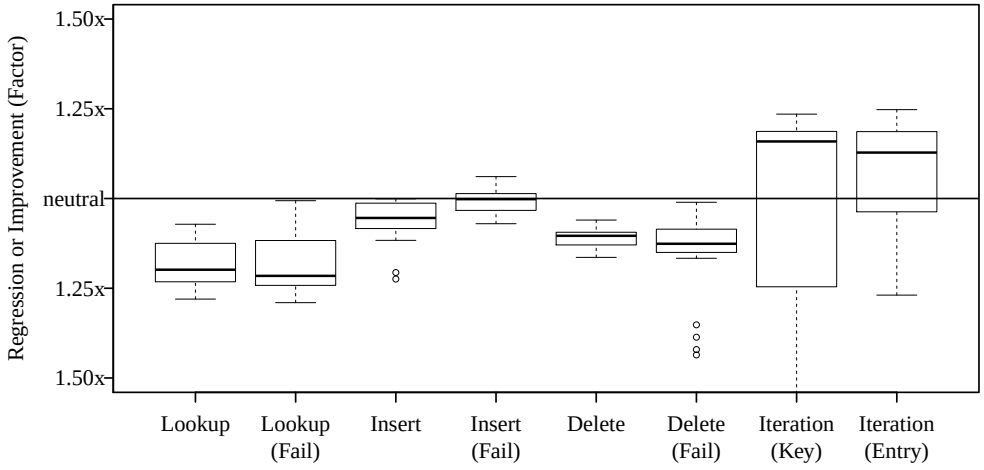
(Partial) Confirmation of Hypothesis 2: Compared to our baseline, and counter to our intuition, HCHAMP improved Iteration (Key) by a median 35 % and Iteration (Entry) by 37 %. The more general HHAMT improved Iteration (Key) by a median 16 % and Iteration (Entry) by 13 %. However, the value spread in Figure 5.1 appears large and the maximum bounds are violated for Iteration (Key).

(Partial) Confirmation of Hypothesis 3: On average, we observed an overhead of 3 % for Lookup and 6 % for Lookup (Fail) when comparing a specialized HHAMT against its regular HHAMT counterpart. These numbers confirm our intuition and are lower then the 20–40 % overhead reported by Steindorfer and Vinju [SV14]. The median costs for Insert (24 %) and Delete (31 %) however match their results. Specializations improved memory consumption by at least 38 % for data structures with 32 or more entries.

Confirmation of Hypothesis 4: Memory footprints of HCHAMP and HHAMT (omitted in Figure 5.1) match exactly CHAMP's, when using multi-maps as maps.



(a) HCHAMP multi-map versus CHAMP map (baseline).



(b) HHAMT multi-map versus CHAMP map (baseline).

Figure 5.1: Visualizing the overhead of various multi-map implementations over a CHAMP map implementation.

Discussion. A more detailed investigation revealed that for Iteration (Key) measurements at sizes 2^1 and 2^5 showed significant deviation from the remaining measurements. These two measurements were not statistically identified as outliers due to the small sample size of 23 (sizes 2^x for $x \in [1, 23]$). When removing these two measurements, the upper bound of slowdowns is 6 % for HHAMT and 36 % for HCHAMP.

While not impacting lookup performance, specializing trades the runtime performance of insertion and deletion for gaining savings of approximately 1.4 x.[§] Because only operations that allocate new tree nodes are affected, we attribute slowdowns to the lookup table we introduced (adding two memory indirection). Nevertheless, specializing is of key importance when optimizations for primitive data types; we evaluate that effect separately in Section 5.6.

Summary. Despite its more complex and heterogeneous encoding, HHAMTs achieves excellent runtimes across all tested operations. Converting a map into a multi-map with the means of a heterogeneous encoding had usually less costs associated than we expected beforehand. Our specialization approach could successfully mitigate overhead for lookups while reducing memory footprints. However, using a lookup table for our specializations still impacts insertion and deletion, when compared to regular array allocations that do not require a lookup table.

5.5 Benchmarks: Comparing Multi-Map Designs

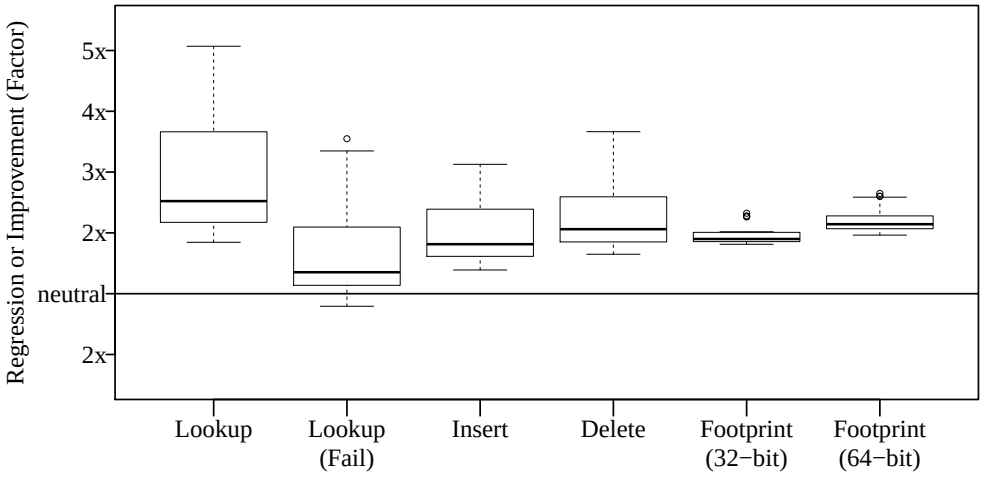
We further evaluate the performance characteristics of our specialized HHAMT multi-map against implementations from Clojure and Scala. Both languages do not provide native immutable multi-maps in their standard libraries, however suggest idiomatic solutions to transform maps with nested sets into multi-maps.

VanderHart [VN14, p. 100–103] proposes a solution for Clojure based on "protocols". Values are stored untyped as either a singleton, or a nested set. Consequently, the protocol extension handles the possible case distinctions —not found, singleton, or nested set— for lookup, insertion, and deletion.

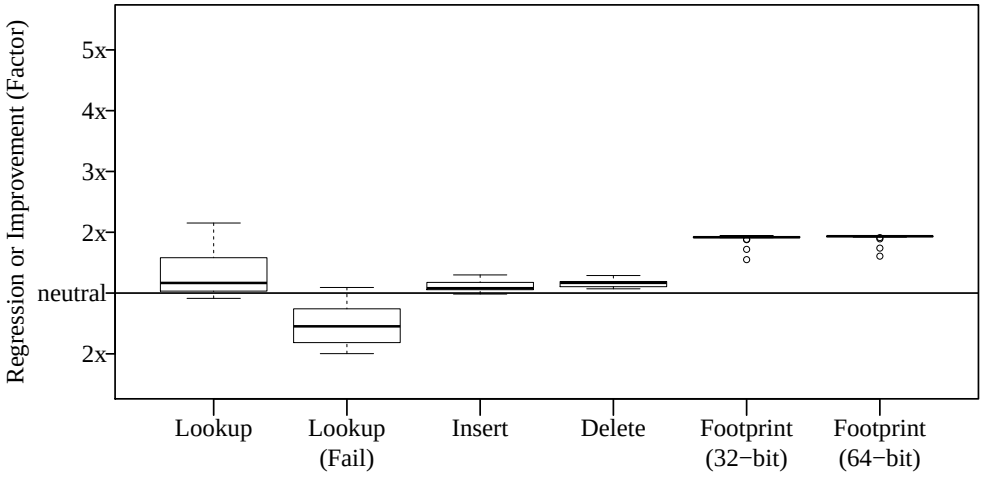
Scala programmers would idiomatically use a trait for hoisting a regular map to a multi-map. However, the Scala standard library only contains a trait for mutable maps; we therefore ported the standard library program logic of the trait to the immutable case, nesting typed sets into maps.

Hypotheses. We expect specialized HHAMT's runtime performance of lookup, deletion, and insertion to equal the competitors performance, because we tried hard to

[§]Note that only the outer multi-map structure was specialized and not the nested sets. A further specialization of the nested sets would yield even more substantial memory savings.



(a) Specialized HHAMT multi-map versus Clojure's multi-map (baseline).



(b) Specialized HHAMT multi-map versus Scala's multi-map (baseline).

Figure 5.2: Performance comparison of a specialized HHAMT multi-map against implementations in Clojure and Scala.

mitigate the incurred overhead, and the idiomatic solutions require some overhead as well. Runtimes should not degrade below a certain threshold —say 10 % for median values and 20 % for maximum values would just be acceptable— (Hypothesis 5). However, for negative lookups we expected that specialized HHAMT performs worse than Scala (Hypothesis 6). This hypothesis is based on related work [SV15] that explains the inherent differences between CHAMP and Scala when it comes to memoizing hash codes. Our hypothesis expects memory improvements by at least 50 % on average due to omitting nested collections for singletons (Hypothesis 7).

Experiment Setup

Data generation is derived from the experimental setup outlined in Section 5.4. We keep the number of unique keys equal —2^x for $x \in [1, 23]$— but instead of using distinct data in each tuple, we now use 50 % of $1:1$ mappings, and 50 % of $1:2$ mappings. Fixing the maximal size of right-hand side of the mapping to 2 may seem artificial, but it allows us to precisely observe the singleton case, the case for introducing the wrapper and the overhead per additionally stored element. The effect of larger value sets on memory usage and time can be inferred from that without the need for additional experiments.

Insert: We call insertion in three bursts, each time with 8 random parameters to exercise different trie paths. 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 canonicalization where applicable.

Lookup: Similar to Delete we call lookup in two bursts to exercise full and partial matches.

Lookup (Fail): In a single burst with 8 random parameters we test negative lookups (neither key nor value present). We assume this test equivalent to Delete with no match.

Experiment Results

Figures 5.2(a) and 5.2(b) show the relative differences of specialized HHAMT multi-map compared to the implementations in Clojure and Scala. From the data we can evaluate our hypotheses:

Confirmation of Hypothesis 5: Runtimes unexpectedly improve over the competition. Lookup, Insert, and Delete perform similar to Scala (by a median 12 %, 9 %, and 16 % faster), and clearly better than Clojure (by a median speedup of 2.51 x, 1.75 x, and 2.05 x). Compared to Scala we observed individual data points that exhibited minimal slowdowns of less than 9 % at larger input sizes.

Confirmation of Hypothesis 6: HHAMT performs worse than Scala for negative lookups. Runtimes increased by a median 39 % and roughly doubled at maximum with a 106 % increase. In contrast, when compared to Clojure we do not see a negative impact.

Confirmation of Hypothesis 7: Memory footprints improve by a median factor of 1.92 x (32-bit) and 1.93 x (64-bit) over the implementation in Scala, and over in Clojure by a median factor of 1.9 x (32-bit) and 2.14 x (64-bit).

Discussion. We were surprised that the memory footprint consumptions of Clojure's and Scala's multi-map implementations are essentially equal. From related work [SV15] we knew the typical trade-offs of both libraries: Scala mainly optimizes for runtime performance, while Clojure optimizes for memory consumption. Code inspection revealed the cause of Scala's improved memory performance: their immutable hash-sets contains a specialization for singleton sets.

All three libraries libraries follow different paradigms for avoiding code duplication in their collection libraries. While Scala and Clojure use extension mechanisms (i.e., traits and protocols respectively), HHAMT avoids duplication by supporting internal heterogeneity.

Summary. With microbenchmarks we were able to measure the performance of individual operation, and further to measure the footprint of synthetically generated structures of different sizes. In this setting the heterogeneous design of specialized HHAMT proved to be better in general: improved runtimes of lookup, insertion, and deletion —with the notable exception of negative lookups when compared to Scala— and most importantly memory improvements of 1.9–2.14 x.

5.6 Case Study: Primitive Collection Footprints

A type-safe heterogeneous HAMT encoding shines most with bounded numerical data: it allows to exploit the difference between primitive (value-based) data types and reference types. More specifically, a `Multimap<int, int>` can leverage storing unboxed inlined singletons. Any non-heterogeneous immutable collection structure would have to store boxed integer objects instead, if not singleton sets of boxed

integers. So, instead, as a fair point-of-reference we will compare to the state-of-the-art hand-optimized specialized immutable data structures for primitive values.

We are not aware of any comparable persistent or immutable primitive collection library which is optimized for primitive data types on the JVM. While there are many specialized primitive collection libraries for the JVM, only some contain (slower) copy-on-write immutable data structures implemented as facades over their mutable counterparts. With respect to primitive multi-maps, we did not find any implementation, neither mutable nor mutable.

So, we concentrate on comparing the memory footprint of `Map<int, int>`, implemented as a specialized HHAMT (with 1 : 1 mappings) compared to the most efficient primitive *mutable* collections we are aware of, namely: Goldman Sachs Collections, FastUtil, Trove, and Mahout. As a point of reference we also include Guava's `RegularImmutableMap` because it is a well-known library, but commonly known to be non-optimal in terms of memory consumption.

Experiment Results

Table 5.6 illustrates observed memory footprints for maps for sizes 2^x for $x \in [1, 23]$. At each size, measurements are normalized with respect to the minimum memory footprint (retained size of heap graph). Consequently, the minimum value 1 depicts the smallest data structure, whereas all other data points are displayed in their relative distance (factor of how much more memory they consume).

The results show that HHAMT consistently consumes the least amount of memory (median 1.00 x, range 1.00–1.10 x), followed by Goldman Sachs (median 1.04 x, range 1.00–2.18 x) and FastUtil (median 1.07 x, range 1.00–4.18 x). Trove exhibits constant results within a small bandwidth (median 1.23 x, range 1.15–2.15 x). In contrast to Trove's constant results, Mahout delivered surprisingly inconsistent results (median 1.94 x, range 1.22–29.64 x). With overheads of 29.64 x, 25.08 x, 19.18 x, 11.24 x and 4.72 x for the data points 2^1–2^5, Mahout exceeds the footprints of our generic reference data structure from Guava (median 4.00 x, range 2.27–4.72 x).

Discussion. Compared to all other primitive collections, HHAMT excelled especially at small collections up to 32 elements. Given that in practice most collections are small [MS07] these improvements look promising. Primitive collections in general have the problem how to mark which slots are in use (there is no `null` equivalent in value types). Several encodings —e.g., sentinel values, or bitmaps— exist to circumvent this limitation. HHAMT performs well with respect to primitive collections, because HHAMT inherently encodes information about the presence and type of (primitive) values on a per node basis an therefore obsoletes special encodings for sentinel values. Further applications and benefits of heterogeneous data structures are discussed in Section 5.9.

Table 5.1: Comparing the memory footprint of a HHAMT map (specialized for **`int`**) to state-of-the-art primitive **`int`** maps. Guava*** is our reference data point for a generic map containing boxed integers, i.e., `Map<Integer, Integer>`. At each size, measurements are normalized with respect to the minimum memory footprint observed (smaller is better).

# Map Entries	HHAMT	Goldman Sachs	Trove	FastUtil	Mahout	Guava***
2^1	1.00	2.18	1.91	4.18	29.64	2.27
2^2	1.00	1.85	2.15	3.54	25.08	3.15
2^3	1.00	1.41	2.06	2.71	19.18	4.29
2^4	1.00	1.38	1.97	1.59	11.24	4.72
2^5	1.00	1.04	1.32	1.13	4.72	3.84
2^6	1.05	1.00	1.25	1.04	2.40	3.83
2^7	1.00	1.04	1.29	1.06	1.28	4.07
2^8	1.00	1.14	1.41	1.15	1.41	4.51
2^9	1.00	1.07	1.22	1.07	1.22	4.24
2^{10}	1.03	1.00	1.15	1.00	1.94	3.99
2^{11}	1.10	1.00	1.15	1.00	1.76	3.99
2^{12}	1.00	1.00	1.15	1.00	1.68	4.00
2^{13}	1.00	1.11	1.27	1.11	1.76	4.43
2^{14}	1.00	1.07	1.23	1.07	1.61	4.29
2^{15}	1.04	1.00	1.15	1.00	1.41	4.00
2^{16}	1.10	1.00	1.15	1.00	1.33	4.00
2^{17}	1.00	1.00	1.15	1.00	1.23	4.00
2^{18}	1.00	1.11	1.27	1.11	1.27	4.43
2^{19}	1.00	1.07	1.23	1.07	2.08	4.29
2^{20}	1.04	1.00	1.15	1.00	1.94	4.00
2^{21}	1.09	1.00	1.15	1.00	1.94	4.00
2^{22}	1.00	1.00	1.15	1.00	1.94	4.00
2^{23}	1.00	1.11	1.28	1.11	2.16	4.44
min	1.00	1.00	1.15	1.00	1.22	2.27
median	**1.00**	**1.04**	**1.23**	**1.07**	**1.94**	**4.00**
max	1.10	2.18	2.15	4.18	29.64	4.72

Summary. In our measurements, HHAMT multi-maps that are specialized for **`int`** consume (with 1 : 1 data) a median 4 x less memory than generic map data structures. HHAMT further achieves the same small footprints as class-leading primitive **`int`** maps, while providing the additional functionality of allowing $1 : n$ mappings.

5.7 Case Study: Static Program Analysis

The above experiments isolate important factors, 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 sets [ASU86]. The nodes in the graphs are complex recursive ASTs with arbitrarily complex (but linear) complexity for `hashCode` and `equals`. More importantly, the effect of the heterogenous 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.

Code. 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-array mapped tried [SV15]. We implemented the following two equations directly on top of the multi-maps:

$$\mathrm{Dom}(n_0) = \{n_0\}$$
$$\mathrm{Dom}(n) = \left(\bigcap_{p \in \mathrm{preds(n)}} \mathrm{Dom}(p) \right) \cup \{n\}$$

Our code* uses set union and intersection in a fixed-point loop: Dom and preds are implemented as multi-maps. The big intersection is not implemented directly, but staged by first producing a set of sets for the predecessors and intersecting the respective sets with each other.

Hypotheses. On the one hand, since Dom is expected to be many-to-many with large value sets it should not generate any space savings but at least it should not degenerate the runtime performance either compared to CHAMP (Hypothesis 8). On the other hand we expect preds to be mostly one-to-one and we should get good benefit from the inlining of singletons (Hypothesis 9). Since CHAMP was reported to outperform existing state-of-the-art implementations in Scala and Clojure on the same case, there is no need to further include these [SV15].

*`http://michael.steindorfer.name/drafts/hamt-heterogeneous-artifact`

Table 5.2: Runtimes of HHAMT for the CFG dominators experiment per CFG count, and data shpae statistics over preds relation (unique keys, tuples, 1 : 1 mappings).

#CFG	CHAMP	HHAMT	#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 %

Data. For our experiment, we used ±5000 control flow graphs for all units of code (function, method and script) of Wordpress,[†] by using the PHP AiR framework [HK14]. Like before, we used JMH to measure CPU 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. Since smaller graphs occur much more frequently, we selected samples with exponentially increasing sizes from 128 to 4096. We furthermore measured the number of many-to-one and many-to-many entries in the preds relation.

Results. The results were obtained with a Linux machine running Fedora 20 (kernel 3.17). It featured 16 GB RAM and an Intel Core i7-2600 CPU with 3.40 GHz (8 MB LLC with 64-byte cache lines). Frequency scaling was disabled.

Table 5.2 shows the mean runtimes of the experiment for CHAMP and HHAMT. Both perform almost identically, with at most ±2 s difference. Due to equal runtimes, HHAMT retains the same magnitude of speedups that CHAMP yielded over Clojure and Scala [SV15], 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. In the case of Wordpress, the CFG algorithm turns out to profit over CHAMP in terms of memory savings from the heterogeneous opimizations for 1 : 1 mappings. We conclude both Hypothesis 8 and 9 to be confirmed.

[†] `https://wordpress.com`

5.8 Related Work

Reducing the Memory Footprint of Collections is a goal of other people as well. Gil et al. [GS12] identified sources of memory inefficiencies in Java's mutable collections and proposed memory compaction techniques to counter them. They improved the memory efficiency of Java's `Hash{Map,Set}` and `Tree{Map,Set}` data structures by 20–77 %. We observed that even with added heterogeneity, HHAMT multi-maps achieve lower memory footprints than the class-leading primitive collection libraries, and in the generic case on average 4 x improvements over Guava's maps.

Steindorfer and Vinju [SV14] 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. In contrast, we targeted the root causes of inefficiency one-by-one allowing full specialization at all arities.

Optimizing Collections in Dynamically-Typed Languages. Runtimes of dynamically typed languages often introduce a significant run-time and memory overhead [Trao9] due to generic collection data structures that could at run-time hold a heterogeneous mix of data.

Bolz et al. [BDT13] introduced a technique dubbed *storage strategies* that enables dynamic conversion of data representations. A set of interlinked strategies form a fine-grained type lattice that is based on known optimizations. Strategies mostly include support for collections of a homogeneous (primitive) type. An exemplary lattices for a `Set` data structure could be `EmptySetStrategy <-> (Integer|Float|...)SetStrategy <-> ObjectSetStrategy`. Resulting performance improvements mainly stem from object layouts that specialize for a homogeneous primitive types and corresponding optimized operations (e.g., direct value comparisons instead of calling `equals` methods).

Bolz [BDT13] showed that with Python on average 10% of collections dehomogenize, mostly at small sizes. These results suggest that even in the absence of strict typing, collections are often used homogeneously. Heterogeneous data structures are orthogonal to homogeneous storage strategies. On one hand, heterogeneous data structures could diversify current strategy approaches, e.g., when homogeneous strategies are not applicable, or when many conversion occur. On the other hand, they have the potential to replace homogeneous strategies when flexibility in mixing data is required upfront. Furthermore, HHAMT optimizes internal heterogeneity that occurs in general purpose data structures such as multi-maps.

Specializations and Generics for Primitives Reducing Code-Bloat. Specializing for primitives can lead to a combinatorial explosion of variants amplifying code-bloat. Due to the object vs. primitive dichotomy, Java does not offer solutions countering a

combinatorial explosion of code duplication when specializing for primitives. Java 10 or later will solve this issue by supporting generics over primitives.*

Ureche et al. [UTO13] presented a compiler-based specialization transformation technique called *miniboxing*. Miniboxing adds automatic specializations for primitive JVM data types to the Scala compiler while reducing the generated bytecode. Combinatorial code-bloat is tackled by specializing for the largest primitive type **`long`**, together with automatic coercion for smaller-sized primitives. While not always memory-optimal due to always utilizing **`long`** variables, miniboxing is a practical approach to tackle combinatorial code explosion.

HHAMT's contribution is orthogonal to both previously discussed techniques, because it generalizes the encoding of heterogeneous data stored together in a collection. HHAMT's specializations currently do duplicate code for different (primitive) type combinations. Using primitive generics in later versions of Java —or miniboxing in Scala— could bring this down to a single generic specialization per trie node arity.

(Partial) Escape Analysis. Escape analysis enables compilers to improve the runtime performance of programs: it determines whether an object is accessible outside its allocating method or thread. Subsequently this information is used to apply optimizations such as stack allocation (in contrast to heap allocation), scalar replacements, lock elision, or region-based memory management [Sta+15]. Current JVMs use partial escape analysis [Sta14], which is a control-flow sensitive and practical variant tailored toward JIT compilers.

Our encoding of specializing is a memory layout optimization for value-type based data types: trie nodes are specialized for arrays of constant sizes that do not escape. We use code generation to conceptually apply object inlining [Wimo8] of statically sized (non-escaping) arrays into the memory layout of their corresponding trie nodes. Memory layout sensitive inlining as we perform could be applied in Virtual Machine (VM) based on information obtained from escape analysis. We hope that future compilers and language runtimes are capable of doing so out-of-the-box.

5.9 Further Applications of Heterogeneous Data Structures

We extrapolate some client applications which would benefit from HHAMT.

Libraries or Languages Supporting Data or Code Analysis on the JVM would benefit from more efficient in-memory multi-maps. Typical examples are frameworks such as KNIME [Ber+09] for general purpose data analysis or Rascal for program analysis [KvdSV09], and MoDisCo [Bru+10] for software re-engineering and reverse

*`http://openjdk.java.net/projects/valhalla/`

engineering, especially when their algorithms require frequent lookup and thus will benefit from an efficiently indexed relation such as a multi-map.

Unifying Primitive and Generic Collections. Looking at specialized collections for primitives from the programming language designer's perspective, they are a necessary evil implied by the dichotomy between objects and primitive values. Primitive values give programmers access to low level and memory-efficient data representations, but the impact of having them leaks through in the type systems and the design of standard libraries of programming languages supporting them. The current chapter describes a heterogeneous framework that can be used for implementing data structure which allow storing either primitive data values or their boxed counterparts next to each other, while the client code remains practically oblivious. For statically-typed languages this implies we can have a generically typed library for both primitive and object values. For dynamically-typed languages it implies a much lower overhead for the builtin dictionaries.

Big Integers for Big Data. Most programming languages feature a library for representing arbitrary-sized integers. We use these to avoid overflow, especially in the context of scientific computing applications. The drawback of using these libraries for data science is that large homogeneous collections immediately blow up, even if the big numbers are exceptional. We want to use smaller FIXNUMs (FIXNUMs) were possible, and BIGNUMs (BIGNUMs) only when necessary.

This application is where HHAMT could potentially have a rather big impact. Sets and maps filled with mostly inlined FIXNUM's and an occasional BIGNUM without having to a priori allocate space for BIGNUMs, and without having to migrate at run-time. Even if the entire collection accidentally ends up filled with BIGNUMs, HHAMT still is more memory efficient than common array-based hash-maps.

Cleaning Raw Data in a Type-Safe Manner. The HHAMT data structure enables efficient storage and retrieval of objects of incomparable types without memory overhead (no need to wrap the objects) and without dynamic type checks. In Java there exist no "union" types like in C, but using HHAMT we can approach this in the context of collections. A typical use case would be reading in raw data from Comma-Separated Values (CSV) files (or spreadsheets) in Java where the data is not cleansed and some cells contain integers while the other contain decimal numbers or even empty cells, depending on the original manual and unvalidated input of the user. A CSV parser could output a HHAMT, inferring the most accurate value for each cell from the used notation, and allowing for further processing the data downstream in a manner both type-safe and efficient.

In general, homogeneous collections storing numeric data struggle with representing empty cells. Sentinel values (e.g., integer constant -1) are a viable solution if and only if the data does not use the data type's full value range. Cases where the data range is used exhaustively require additional auxiliary data structure (e.g., an array of booleans) to encode if a value is initialized. In contrast to homogeneous collections, HHAMTs by design supports mixing sentinel values of a different type (e.g., `static final EMPTY_CELL = new Object()`) with the full value range of primitives.

5.10 Conclusion

We proposed HHAMT, a new design for hash-array mapped tries which allows storage, retrieval and iteration over maps which store heterogeneously typed data. In particular we motivate this data structure by applying it to efficiently implement multi-maps, and it also shines when used to specialize for primitive types.

The evaluation compared to the state-of-the-art: comparing to other hash-trie data structures with and without the many-to-many feature, comparing against state-of-the-art encodings of multi-maps in Scala and Clojure and comparing to hand-optimized maps for primitive values. Even when compared unfairly to implementations which do not feature heterogeneity, HHAMT compares well. We safe a lot of memory (2–4 x) at relatively low costs in runtime overhead.

We hope multi-maps based on these results will be available in the future in the standard libraries for collections on the JVM, since that would make the JVM even more attractive for safely computing with large immutable datasets.

Chapter 6

Performance Modeling of Maximal Sharing

It is noticeably hard to predict the effect of optimization strategies in Java without implementing them. "Maximal sharing" (a.k.a. "hash-consing") is one of these strategies that may have great benefit in terms of time and space, or may have detrimental overhead. It all depends on the redundancy of data and the use of equality.

We used a combination of new techniques to predict the impact of maximal sharing on existing code: Object Redundancy Profiling (ORP) to model the effect on memory when sharing all immutable objects, and Equals-Call Profiling (ECP) to reason about how removing redundancy impacts runtime performance. With comparatively low effort, using the MAximal SHaring Oracle (MASHO), a prototype profiler based on ORP and ECP, we can uncover optimization opportunities that otherwise would remain hidden.

This is an experience report on applying MASHO to real and complex case: we conclude that ORP and ECP combined can accurately predict gains and losses of maximal sharing, and also that (by isolating variables) a cheap predictive model can sometimes provide more accurate information than an expensive experiments.

This chapter is based on the following published article: Michael J. Steindorfer and Jurgen J. Vinju. 2016. Performance Modeling of Maximal Sharing. In Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering (ICPE '16). ACM, New York, NY, USA. URL http://dx.doi.org/10.1145/2851553.2851566.

6.1 Introduction

This section is concerned with performance modeling of Java libraries. "Premature optimization is the root of all evil", says Donald Knuth [Knu79]. The reason is that optimization strategies are prone to make code more complex and perhaps for no good reason because they may backfire unexpectedly.

Our question is: how can we know, a priori, that a particular optimization strategy will pay off? For most optimizations there is only one way to find out: implement an optimization and compare runtime characteristics against an unoptimized version. In reality it will often take multiple rounds of profiling and tuning before the desired effect and the promised benefit of an optimization is attained. In this chapter we present the MAximal SHaring Oracle (MASHO): a prototype profiling tool that predicts the effect of the maximal sharing optimization a priori, avoiding costly and risky engineering. We report on the experience of testing MASHO and trying it out on a real and complex case.

The "maximal sharing" optimization tactic, dubbed "hash-consing" [Got74], entails that selected objects that are equal are not present in memory more than once at a given point in time. To make this happen a global cache is used to administrate the current universe of live objects, against which every new object is tested. There are two main expected benefits of maximal sharing: avoiding all redundancy by eliminating clones in memory, and the ability to use constant time reference comparisons instead of deep `equals` checks that are in $O(\textit{size of object graph})$. This is because maximal sharing enforces the following invariant among selected objects: $\forall\ \textit{objects}\ x, y : x.\textit{equals}(y) \Leftrightarrow x == y$, which allows any call to `equals` on shared objects to be replaced with a reference comparison. The expected overhead is the maintenance of a global cache, and for each object allocation, extra calls to the `hashcode` and `equals` methods.

Figure 6.1 illustrates the effect of maximal sharing on an object that is "embarrassingly redundant": a reduction from exponential to linear size (in the depth of the tree). In contrast, a tree with the same structure but all unique integer values in its leaf nodes would have no sharing potential.

Maximal sharing is associated with immutable data structures [Oka99], since it requires objects to not change after allocation. It is applied in the context of language runtimes of functional languages [Vis04; KvdSV09], proof assistants [BJM13], and algebraic specification formalisms [vdBra+02; Bor+98; Cla+01; Mos04; Bal+07], compilers [Vaz+07], or libraries that supply similar functionality. Especially when many incremental updates are expected during computation (i.e., creating redundancy over time) we may expect big benefits. E.g., implementations of term rewriting (reducing trees), type constraint solving (minimizing sets of solutions) and solving data flow equations (incrementally adding/removing graph edges) share these characteristics.

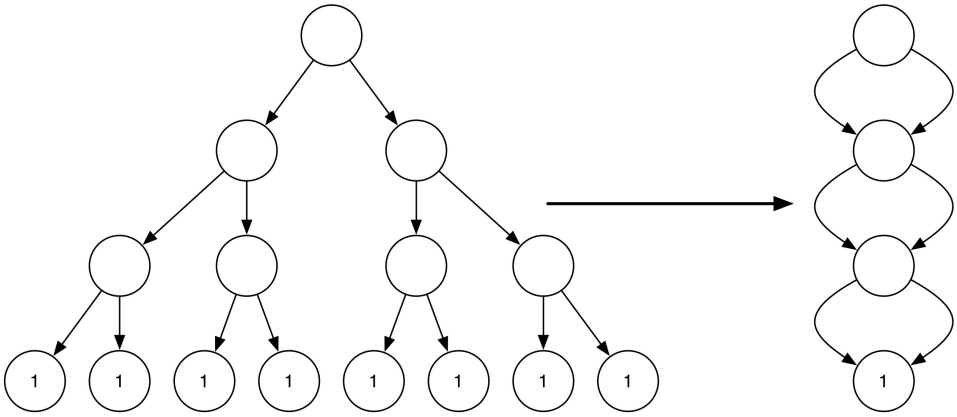


Figure 6.1: Good conditions for sharing: redundant objects.

On the one hand, in the case of high performance implementations of term rewriting engine libraries, maximal sharing has proven to be a very successful strategy: *"It turns out the increased term construction time is more than compensated for by fast equality checking and less use of space (and hence time)"* [vdBra+02]. In real cases memory savings between 52.20%–98.50% of term representations were observed [vdBK07]. On the other hand, maximal sharing can have a negative net effect on memory consumption in absence of enough redundancy, due to the overhead of the global cache.

The audience for the maximal sharing technique is library developers rather than application developers. Considering the effort associated with optimizing for maximal sharing, an often reused library is expected to have larger return on investment than a single application.

With the advance of immutable objects and functional language constructs in object-oriented languages —like Java 8 or Scala, and functional languages running on the Java Virtual Machine (JVM) like Clojure— it is now relevant to investigate if and how we can use maximal sharing to our benefit in JVM library implementations. Immutability may be a too strong requirement for any Java library in general, but if immutability comes naturally for different reasons, then the maximal sharing strategy is an important one to consider.

Potential adopters of maximal sharing suffer from a common problem: converting a library to use maximal sharing is hard and costly [vdBra+00; vdBK07; vdBMV05]: it is a cross-cutting design decision with difficult to tune implementation details. To illustrate one of many pitfalls, let us take a Java library for graph processing as example. It makes use of standard library classes for integers and sets. The

hashcodes of empty sets are 0 in Java and for singleton sets the hashcodes are equal to the hashcode of the elements, because `hashCode()` of a `java.util.Set` is defined to be the sum of the hash codes of the elements in the set. Inserting such similar values in a global cache for sharing would trigger unexpected hash collisions. The success of maximal sharing depends on one hand on a broad spectrum of properties of a library, like its Application Program Interface (API) design, quality of hash codes, co-optimization of shared data structures, and on the other hand on runtime characteristics like data redundancy and the ratio between object allocations and equality checks of shared objects. Naive implementations of maximal sharing —that do not take these issues into account— are likely to slow down programs and increase their memory footprint.

Contributions and Outline

This chapter does not contain an evaluation of the maximal sharing optimization technique; it does contain an evaluation of the accuracy and usefulness of a modeling and simulating technique for maximal sharing. The contribution of this chapter is firstly the design of MASHO (Section 6.2), which includes:

- Object Redundancy Profiling (ORP): measuring the lifetime of redundant objects during a program execution, optimized to benefit from immutable data and to include the notion of data abstraction to accurately model the possible effects of maximal sharing;
- Equals-Call Profiling (ECP): capturing the recursive call-graph shapes of calls to `equals`, including a partial alias analysis;
- Maximal Sharing Model (MSM): a lightweight predictive model that uses ORP and ECP profiles to predict the behavior of a program after the application of maximal sharing.

Secondly we contribute an experience report on the use MASHO for modelling the runtime environment of a programming language. From this we learned that:

- it predicts the impact of maximal sharing on memory usage and the use of equality very accurately and so it removes the need for direct experimentation with maximal sharing, producing equivalent information for making go/no-go decisions with a mean slowdown of 7x for ORP and ECP;
- it isolates the effects of introducing maximal sharing from the effects of JVM configuration (e.g., memory bounds, garbage collector heuristics) and accidental hash collisions that would occur due to a global cache.

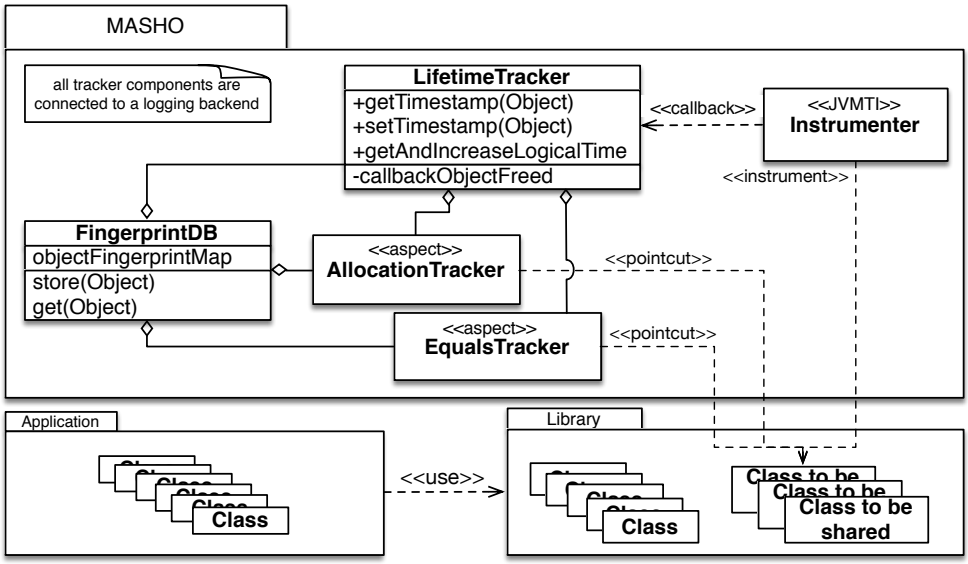


Figure 6.2: Class diagram and Aspect-Oriented Programming profile depicting MASHO's architecture.

- for the validating experiment, a set of realistic demonstrations, implementing maximal sharing will produce good memory savings but will not lead to performance speed-up without first applying major changes to the semantics of the library. We can decide a "no-go".

In general, this experience shows how cheap predictive performance modelling can produce more actionable information than an expensive real-world experiment can since it can soundly factor our confounding factors like the Java garbage collector and reason about otherwise infeasible design alternatives. Related work is further discussed in Section 6.8, before we summarize in Section 6.9.

6.2 Architecture and Design Decisions

In the following we describe the design decisions and most important implementation details of MASHO in this order: how it is used by a library developer, its architecture and implementation choices, and what its preconditions are.

Library Developer Perspective

The user first configures MASHO with a list of interesting classes or interfaces which might hypothetically benefit from maximal sharing. MASHO then instruments the library (which does not implement maximal sharing). Next, the user runs programs that use the library, while the instrumentation logs information to be analyzed. After this, MASHO analyzes the logs producing charts and tables explaining the likely effect of maximal sharing on the library in the context of the executed programs. The user interprets the charts to decide go/no-go on investing in maximal sharing, or continues tweaking the experiment's setup or the test runs.

Instrumenting a Program for Profiling

Figure 6.2 depicts the architecture of MASHO. A client library is instrumented using both AspectJ and the Java Virtual Machine Tool Interface (JVMTI) for gathering the following events: object allocations, object garbage collections, and calls to `equals`. An AspectJ pointcut selects all constructor call-sites of to-be-shared classes. In an advice, which gets called whenever one of these constructors executes, MASHO performs at run-time ORP with fingerprinting (Section 6.3) and an alias-aware object graph size measurement (Section 6.3). Similarly, we use pointcuts for ECP to record the call-graph shape of `equals`-calls (Section 6.3). Bytecode Instrumentation (BCI) is used to track object allocations that are otherwise intangible for AspectJ, for example object construction via reflection. For lifetime tracking, we tag each newly allocated object with the aid of the JVMTI to get notified about an object's garbage collection.

The Precondition: Weak Immutability

Maximal sharing introduces a global cache for all designated objects and uses an object factory for creating new instances. Instead of `new Tuple(a, b)`, one would call a factory method like so: `factory.tuple(a, b)`. Whenever the factory encounters that an equal object already exists in the global cache, it returns the cached instance and forgets the temporary object. Otherwise it caches the new instance and returns it. Such a global cache introduces data dependencies between parts of the program which would normally be unrelated to each other. Consequently, maximal sharing does not work for mutable objects because it may break referential integrity: if a shared object would change, this would become observable in otherwise logically independent parts of the program.

One way to avoid breaking programs in the presence of maximal sharing is requiring full immutability of shared objects, but such a strong condition is not necessary. Therefore we define *weak immutability*, a sufficient condition under which maximal sharing can work, as follows: for any object o and its updated future value o' it holds that o.`equals`(o'), while observing that not necessarily all fields have to

contribute to its `equals` method. Based on weak immutability, object identity can be defined by the transitive closure of immutable attributes of an object, also known as structural equality [Bak93]. Similarly it follows that all object graphs generated from these classes, if we follow only references to fields that are used by the `equals` methods, are Directed Acyclic Graphs (DAGs).

Competitive tools [MO03] solely reason on the granularity of "physical" object-graph equality, while logical object equality may need some form of abstraction. For example, in case of unordered collection data structures such as hashtables, and lazily instantiated (caching) fields. We will detail in the next section, how to support those cases for better coverage.

6.3 Profiling Techniques

The following section describes how ORP and ECP work as individual techniques. ORP and ECP form the basis of MSM (cf. Section 6.4), which predicts the effect of introducing maximal sharing. We identify possible sources of inaccuracy throughout the text and evaluate these in Section 6.5.

Object Redundancy Profiling

The ORP part of MASHO takes advantage of the weak immutability of the selected classes and the fact that we are guaranteed to analyze data that could be represented as a DAG. Namely, we compute a fingerprint for each object, representing the structural identity of its value, using a bottom-up analysis of the object DAG. Fingerprinting allows us to avoid fully serializing heap objects or logging all changes of the heap to disk [SHM08; MO03]. Instead we serialize only the fingerprints that are expected to be a lot smaller in size.

The fingerprint function f, a cryptographic 256-bit SHA-2 hash function in our case, has similar goals as the normal standard 32-bit integer `hashcode` method but necessarily it has a much higher resolution to better represent object identity. For an optimal f (i.e., perfect hashing) it can be said that for any two objects o_1, o_2 it holds that $o_1.f() = o_2.f() \Leftrightarrow o_1.\texttt{equals}(o_2)$. The inevitable non-optimality of a cryptographic f may introduce inaccuracy in MASHO's profiles, while at the same time making the analysis feasible by avoiding a full serialization of every object.

Weakly-immutable object DAGs can only be created bottom-up, so MASHO computes a fingerprint at each allocation of a to-be-shared object. We use a fingerprint cache to efficiently refer to the fingerprints of already known objects. Therefore, fingerprinting a new composite object is always $O(\textit{shallow object size})$. We distinguish the following cases:

Leaf Objects: are objects that have no children in the immutable DAG. We serialize leaf objects and fingerprint them by applying f on the resulting byte-arrays.

Ordered Composite Objects: are objects that contain an ordered sequence of references to other shared objects. We first lookup and concatenate the fingerprints of all referenced shared objects. Then, we compute f over the concatenated hashes.

Unordered Composite Objects: are objects that contain an unordered sequence of references to other shared objects. We first lookup and concatenate the fingerprints of all referenced shared objects. Then, we reduce the set of fingerprints to a single fingerprint with the bitwise XOR operator. This commutative fingerprint computation is stable under arbitrary orderings.

In the case of unordered composite objects, arbitrary orderings of arrays, containing the same values, are to be expected for example in array-based implementations of hash-maps and hash-sets. We abstract from these arbitrary orderings in order to predict more opportunities for maximal sharing, as well as abstracting away from differences that are due to hash collision resolution tactics.

Object Graph Size Calculation

Modeling memory savings requires reasoning over which references already point to the same objects and which do not. Such aliasing is likely present in any Java application. MASHO computes the memory footprint of a to-be-shared object efficiently at object allocation time, which is sufficient only due to weak immutability. It uses Java reflection to collect fields and compute the size of all referenced objects and contained primitive values. This traversal skips nested to-be-shared objects to solely measure the overhead incurred by redundant objects. Aliases of not to-be-shared objects are detected by maintaining a lookup table that maps object identities to their memory footprints. If an object reference is already present in the table, then we have detected an alias and should not count the same object again, but simply add the size of the 32 or 64-bit reference. Note that this alias analysis is *incomplete* by design due to efficiency considerations. We distinguish two cases: *visible* and *invisible* aliases. While the former is traced accurately, the latter may introduce inaccuracy because we only partly track the heap.

Visible aliases

Visible aliases are references that are reachable from a to-be-shared object. For example, consider two different Java fragments which construct a tuple and its content, an atom. Both classes are to be maximally shared:

- Tuple elements are aliases:
  `Atom a = new Atom("S"); new Tuple(a, a);`
- Tuple elements are unique:
  `new Tuple(new Atom("S"); new Atom("S"));`

ORP should predict savings for the latter because it uses duplicates, whereas the former already shares the atom.

Invisible aliases

Invisible aliases are references to library objects that are outside the interfaces that the library developer chose to track. Consider the following Java fragment: `Atom atom(String s) { return new Atom(s); }`. `Atom` is to be shared, whereas `String` is not. We attribute the size of `s` to the size of the first tracked object that references `s`. Note that `s` might be referenced by any other object: either from an object to be shared, or from an object that is not meant to be shared. The accuracy of MASHO is influenced by this effect (addressed by one of our evaluation questions in Section 6.5).

Equals-Call Profiling

The goal of ECP is to record the shape of (recursive) calls to `equals` on to-be-shared objects. Tracking the calls to `equals` requires detailed consideration to be applicable to maximal sharing. After objects are maximally shared, all calls to `equals` can be replaced by reference comparisons, but also already existing aliases have to be taken into account to not over-approximate the potential benefits of maximal sharing.

In Java it is common that `equals` implementations first check for reference equality on both arguments to short-circuit the recursion in case of aliased arguments. Using AspectJ we cannot easily capture the use of == or !=, but we can measure the difference between root calls to `equals` and recursively nested calls to `equals`. By root calls we mean invocations of `equals` that are not nested in another `equals`-call. In case `equals` implementations do not return on aliases directly, MASHO pinpoints these optimization opportunities by warning the user about them.

6.4 Predictive Modeling of Maximal Sharing

By combining the results from redundancy and `equals`-call profiling, we are able to hypothetically model the impact of maximal sharing, including changes to the heap structure, overhead introduced by a global cache, and substitutions of (recursive) `equals`-calls by reference comparisons.

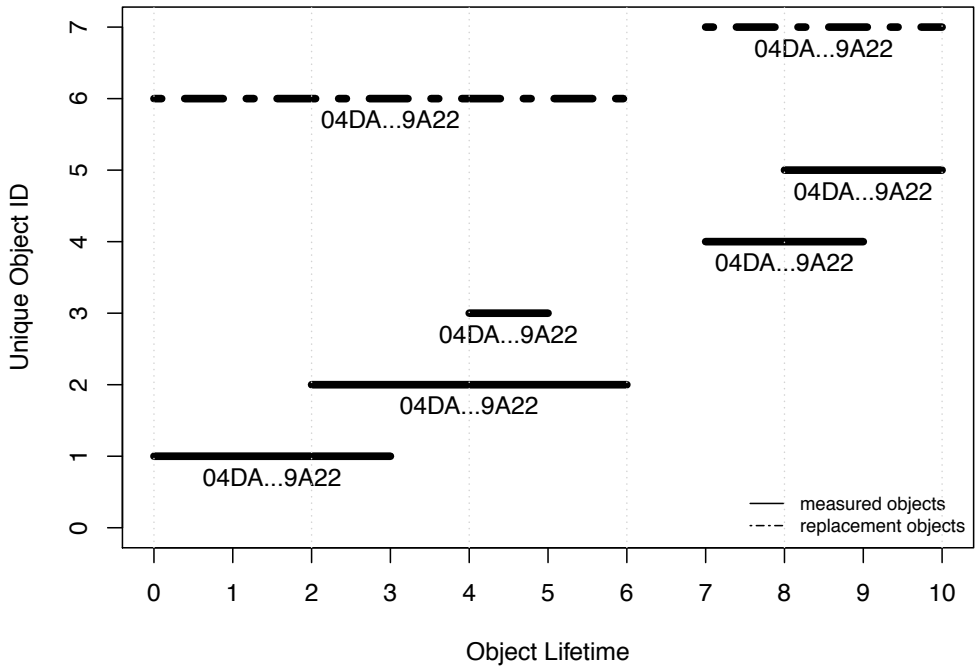


Figure 6.3: Overlapping lifetimes for objects with identical fingerprints. Two families of redundant alive objects are visible (solid lines) and also their possible replacement maximally shared alternatives (dashed lines).

Modeling memory usage. MASHO analyzes the profiled data as follows. It reasons about redundancy that is present at each point in time. Time is measured by allocation and deallocation event timestamps. Figure 6.3 illustrates several objects that map to the same fingerprint *04DA...9A22*. The objects with identifiers 1, 2, and 3 consecutively overlap, as well as objects 4 and 5. We call a sequence of overlapping lifetimes an "object family". In an optimal situation each family would reduce to a single object, with an extended lifetime (see dashed lines in Figure 6.3).

First, we replay the trace to compute the current memory usage of the profiled program for each point in time. We start from a set of object lifetime triples, $allocationTime \times deallocationTime \times uniqueSize$. We compute two lists for these objects: one sorted by allocation time, and another one sorted by deallocation time. Then we traverse the two sorted lists and compute their running sums. At each timestamp the difference between the running sums denotes the current memory usage.

Second, we compute an estimated memory profile of the same program run as-if objects would be maximally shared. Again, we sort the aforementioned object

lifetime triples on timestamps but now we also group them by their fingerprints. This artificially removes duplicate objects and extends the lifetimes of the remaining objects. The final memory usage at each point in time is computed exactly as before, but on this filtered and re-ordered set of lifetime triples. This computation predicts what memory is theoretically minimally necessary to store the observed objects. In practice of course more memory will be used because objects are now even more unlikely to be garbage collected immediately after they become unreferenced. This effect will be observable in the evaluation later.

Modeling the global cache memory overhead. MASHO assumes a fixed bytes-per-record overhead per object reference stored in the global cache that is to be introduced. Predicting the overhead is a matter of multiplying a constant —currently 42— by the number of unique objects at any point in time. To choose its default value, we analyzed the memory overhead of an object repository that is implementable with the standard Java collections API (i.e., `WeakHashMap` with `WeakReferences` as values) and an existing and thoroughly optimized implementation from the ATerm Java library [vdBra+00]. ATerm's global cache imposes a 42 bytes-per-record memory overhead, while a standard `WeakHashMap` implementation requires 79 bytes-per-record.

Modeling the global cache runtime overhead. The expected number of newly introduced calls to the `equals` method is exactly equal to the number of redundant object allocations. The new implementation of `equals` will not have to be recursive anymore under the assumption of maximal sharing. Note that these predictions are under the assumption of optimal hash code implementations and a collision free global cache implementation.

We may also predict the maximal number of new executions of == by counting at each call to `equals` the number of immutable fields, i.e., the arity, of the object. Note that this arity depends on the definition of the original `equals` method. This is the number of fields that contribute to its implementation.

Suppose an implementation of a to-be-shared class uses arrays for storing nested objects. In this case the arity of the object is open and `equals` is in principle in $O(arity)$ even after introducing maximal sharing. The higher this arity, the lower the benefit of maximal sharing will be. This is why MASHO reports also the expected number of newly introduced reference comparisons to the library engineer.

6.5 Evaluation: Setup

Does MASHO allow library engineers to model and simulate what they might get out of maximal sharing without actually implementing and tuning it? Our evaluation questions are:

Q-Accurate: does MASHO predict memory gains and the effect on `equals`-calls after maximal sharing accurately?

Q-Actionable: does MASHO give a library engineer enough information to decide upon further time investments in the maximal sharing strategy?

First, we set up a controlled experiment (Section 6.6) where we can theoretically explain the shape of the input and the shape of the resulting statistics. This is to test whether the experimental setup works reliably and accurately.

To answer **Q-Accurate** we will then compare MASHO's models to profiles obtained from realistic cases that implement maximal sharing (Section 6.7). The hypothesis is that the memory and `equals`-calls models are very accurate, i.e., within a 1% margin of error. The hypothesis assumes that the introduction of the global cache —that holds weak references to shared objects— does not (or only marginally) change the overlapping lifetimes of the object families; we will report whether or assumption holds. The output of these realistic experiments is discussed in detail as a prerequisite to assess **Q-Actionable** qualitatively.

Experience: the Program Data Base Case

We report on our experience testing and evaluating MASHO on two open-source projects: the Program Data Base (PDB),* a library for representing immutable facts about programs, and Rascal [KvdSV09], a Domain-Specific Language (DSL) for meta-programming. Both projects are actively developed and maintained since 2008. Rascal has 110K Source Lines of Code (SLOC) and PDB 23K SLOC. PDB is the run-time system of Rascal. All values produced and consumed by Rascal programs are instances of PDB library classes. This ranges from primitive data types, to collections, and more complex compositional data structures like arbitrary Algebraic Data Types (ADTs). PDB's basic primitive is the `IValue` interface, which every weakly-immutable data type adheres to. Thus, analyzing the usage of `IValue` in Rascal programs is comparable to analyzing how `Object` is used in Java. For the experiments below, we configured MASHO to share all objects of library classes that implement `IValue`.

The PDB classes support two forms of equality. This is common for weakly-immutable libraries with structural equality (cf. Clojure). One is implemented by `equals` satisfying weak immutability by implementing strict structural equality. The other is called `isEqual` and ignores so called "annotations". Annotations are extensions to a data value which should not break the semantics of existing code that does not know about them. The `isEqual` method does satisfy weak immutability, but if maximal sharing would be applied based on the `isEqual` semantics instead of on

*`https://github.com/cwi-swat/pdb.values`

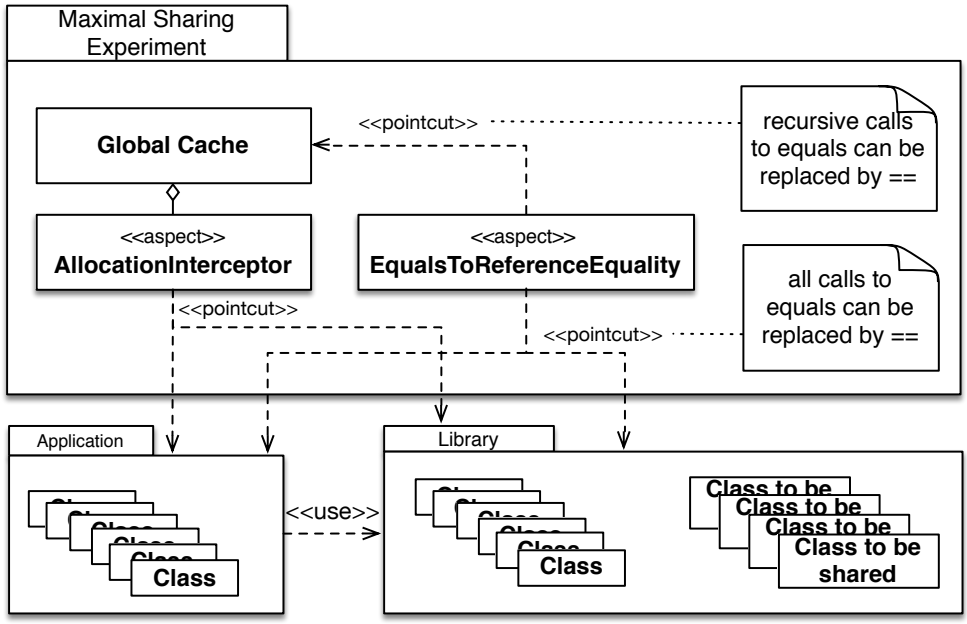


Figure 6.4: Using AspectJ to experiment with maximal sharing.

the `equals` semantics it could break client code: annotations would quasi-randomly disappear due to accidental order of storing (un)annotated values in the global cache. With maximal sharing in mind, annotations should not be ignored for equality.

To be able to analyse `isEqual` as well, we configured ECP to track `isEqual` calls like it tracks `equals`-calls. Note that `equals` and `isEqual` do not call each other. Furthermore, we inserted an additional rule that checks at run-time if both arguments to `isEqual` map to the same fingerprint, by performing fingerprint cache lookups. If yes, we imply that they are strictly equal to each other and do not contain annotations in their object graphs. As a consequence we model such an `isEqual` call as a reference equality, because both arguments will eventually become aliases under maximal sharing. Otherwise, if fingerprints do not match, we continue to recursively descent into the `isEqual` call.

To conclude, PDB represents a well-suited but challenging case for maximal sharing: it is not set up for tautological conclusions of our evaluation of MASHO.

A Minimal Maximal Sharing Realization

For answering the **Q-Accurate** question we should verify the predictive model of MASHO against actual data from a real maximal sharing implementation. To gather memory data and to profile `equals`-calls, a fully optimized implementation is not necessary, and also absolute numbers about runtime performance are not comparable anyway due to interference of the profiler and JVM configuration. Therefore, we should abstract from absolute runtime numbers and instead evaluate the absolute reductions/increases in terms of `equals`-calls.

Figure 6.4 shows a class diagram of how we used AspectJ to obtain a maximally shared version of PDB. Our global cache is implemented using a `WeakHashMap` with `WeakReference` values. We use a pointcut with *around*-advises to intercept and instrument object allocation call-sites (both in library and in application code), substituting new objects with references from the global cache, if present. We also replace all calls to `equals` outside of the central repository by reference equalities, as well as all recursive calls to `equals` called by the central repository. The recursive calls can be replaced because nested values have already been maximally shared.

Shared versus Non-Shared Measurement

We reuse MASHO's measurement facilities to measure both a shared and a non-shared version of each experimental run. In the shared version we reuse a strict subset of MASHO's measurement facilities, namely for ECP, object size calculation, and timestamping of allocation and deallocation events. The latter are services of the JVMTI. Reuse of MASHO's measurement facilities entails a threat-to-validity that is mitigated by the controlled experiment, where we can theoretically explain the shape of the data and the expected outcome. We do completely turn off ORP profiling for these experiments to avoid interference. In the presence of the maximal sharing aspect we can observe real global cache behavior and identify redundant objects based on cache hit counts.

Our first naive memory measurement method (*method 1*) is to aggregate and compare the mean memory usage from a shared library experiment against the model that is calculated from the non-shared profiles. If the difference is small, then MASHO predicts accurately. Otherwise, either the evaluation method is flawed, fingerprinting has too many collisions, or MASHO misses an important aspect in modeling maximal sharing. The hypothesis is that based on this analysis we will see only minor differences because MASHO is expected to produce an accurate model.

The previous method is naive, because we know that the Garbage Collector (GC) will influence the mean memory usage as often and perhaps as much as the optimization strategy does. It is a confounding factor. We should expect sawtooth patterns in MASHO's memory profiles caused by short-living temporary objects that

could all be discarded immediately after allocation —in case of a hit in the global cache— but instead will remain in memory until the GC starts to operate. So, from the comparison of mean memory usage we should hypothesize significant inaccuracy in predicting memory usage.

To mitigate the confounding factor introduced by the delays in garbage collection we may set the heap size setting of the JVM to a benchmark specific global minimum.[†] This would trigger the GC more often and force it to collect temporary objects. The mean memory usage then starts approaching the global minimum in memory usage. While identifying globally minimal heap sizes per benchmark could be automated with bisection search, we argue it is not precise enough for our memory measurements. Therefore we also set up a second method of comparison (*method 2*). This method is similar to the previous, but additionally we tag all short-living temporary objects —by measuring whether they cause cache hits— and subtract their sizes from the actual memory usage. The effect is that we filter noise introduced by the GC. Instead of only considering one global minimum, we now reason over a series of local minima in time. If the difference in memory usage between this minimal size and the predicted size is still large, then MASHO is inaccurate, as caused by fingerprint collisions or an unsound modeling. Otherwise it is accurate. For the sake of transparency we will discuss both the results of the naive method and the mitigated method of comparison.

Setup of JVM Parameters. We use AspectJ 1.7.3 a 64-bit JVM from Oracle, Java version 1.7.0_51, running on an Intel Core i7 3720QM CPU under Mac OS X.[‡] We configured the JVM with the following settings additional to the `-server` flag: with `-Xms4G -Xmx4G` we set the heap to a fixed size of 4GB and prohibit resizing; `-XX:+DisableExplicitGC` deactivates manual invocation of the GC; `-XX:+UseParallelOldGC` uses a parallel collector for new and old generations.

6.6 Evaluation: Controlled Experiment

Here we test-drive our evaluation method. We use two scenarios that are based on the introductory example from Figure 6.1: with the PDB library we first build binary trees of depth d where all leaf nodes are equal with respect to each other, and second binary trees of depth d where all leafs nodes are different from each other. We hypothesize the results of the experiment and observe whether or not these

[†] Minimum heap size is a function of time: each program state has its own minimum. With global minimum we refer to the maximum of all minima, i.e., the lower memory bound that is sufficient to run the program.

[‡] At the time of performing our experiments, the latest stable AspectJ Development Tools (AJDT) version included AspectJ 1.7.3, which only supported Java Development Kit (JDK) 7.

numbers are produced. This check of both an optimal case and a worst case scenario for maximal sharing would reveal obvious problems with our measurements.

Expectations

While profiling we expect from our setup that no object is garbage collected until the program ends and that both trees consume the same heap size. Zero redundancy should be measured in the redundancy-free case, and for depth d in the redundant case $2^{d+1} - d$ duplicates. When running PDB with the maximal sharing aspect, memory savings should be visible for the redundant case, and growing with increasing depth. The controlled experiment only allocates objects, but does not invoke `equals`. However the maximal sharing introduces `equals`-calls by performing global cache lookups. We expect one `equals`-call per cache hit, and furthermore for each binary tree node two reference comparisons, one for the left and one for the right subtree.

Results

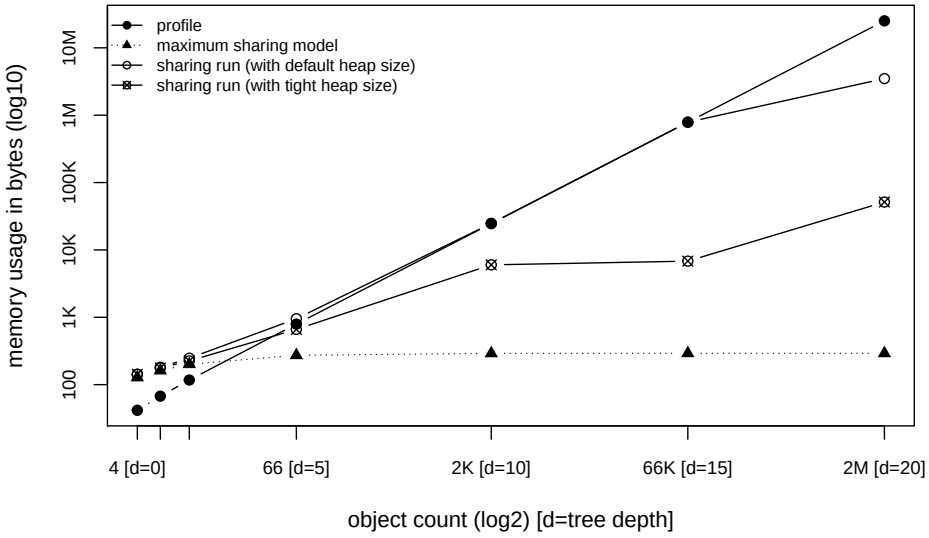
Figure 6.5 shows the results of profiling the creation of trees with depths from 1 to 20. The plots use logarithmic scales.

Redundant Trees

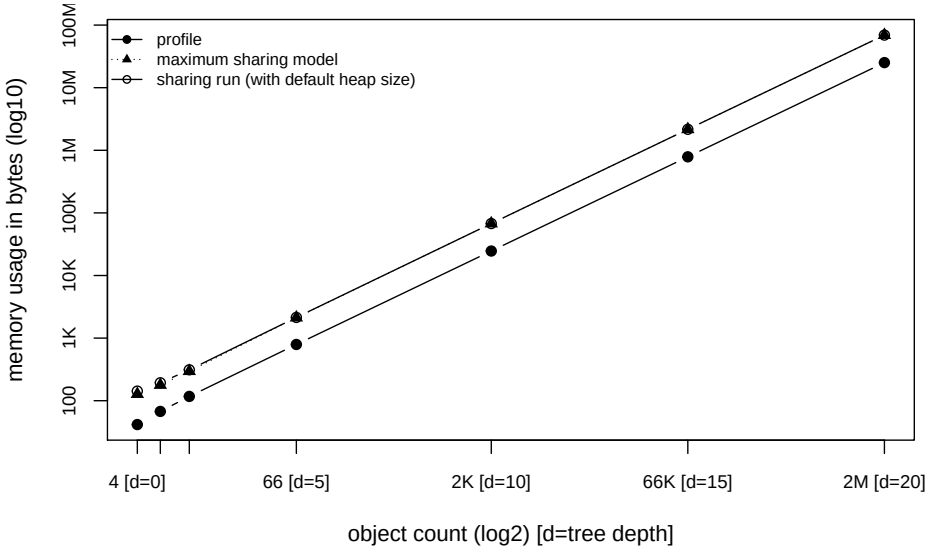
Figure 6.5(a) focuses on redundant trees. The x-axis highlights the profiled allocation count at each depth d. Surprisingly, the measurement at $d = 0$ exhibits four allocations instead of the one expected: Manual inspection revealed that PDB's integer implementation contains an `integer(1)` constant, and further, the two boolean constants `TRUE` and `FALSE` were pre-initialized by the library.

The *profile* line shows memory usage obtained by the profiles, while the *maximum sharing model* line illustrates the predicted minimal heap usage under maximal sharing. At low object counts ($d <= 2$) the *maximum sharing model* signals a higher memory usage with maximal sharing than without, However, at $d = 5$ the measurements break even, denoting a saving potential of 66%. The saving potential stabilizes around 100% from $d = 10$.

The *sharing run (with default heap size)* line shows the heap profile with the maximal sharing aspect applied. For $d < 20$ there is no measurable difference from the *profile* line. Only at $d = 20$ with about 2M object allocations, we see a difference because temporary objects are partially collected. Performing another *sharing run (with tight heap size)*, yields results that are clearly different from the original memory profile, yet a significant error remains. The results confirm that the GC largely influences the naive *method 1* (mean accuracy of 27%, range of 3–93%).



(a) results for redundant trees



(b) results for redundancy-free trees

Figure 6.5: Calibration data: Memory usage for various test runs (without compensation for GC noise). Figure 6.5(a) illustrates that compensating for GC noise is necessary to obtain accurate memory footprint models.

Measurements with *method 2* are not visible in the graph, because the data aligns exactly with our *maximum sharing model*. It performed with 100% accuracy at experiments with an allocation count bigger than 66; at smaller counts the three unexpected allocations reduce accuracy marginally.

The measured global cache hits (that are not listed here for brevity) are exactly off by one due to the `integer(1)` constant. Measured `equals`-calls that are caused by the global cache match exactly with the number of cache hits, as expected. Estimated reference equalities are also accurate: each cache hit of a tree node object yields two reference comparisons, one for each sub-node.

Redundancy-free Trees

Figure 6.5(b) shows the results for trees with no shareable data. The *maximum sharing model* and *sharing run (with default heap size)* correlate. The plot illustrates the overhead of the global cache that grows linearly with each unique object. The only unexpected observation is one additional cache hit, caused by the previously mentioned `integer(1)` constant.

No hash collisions were recorded due to global cache lookups, with the exception of a single experiment ($d = 20$) that yielded 420 `false` equality comparisons in a cache with 2M cached objects. We list the number of equality checks that yielded `false` rather than full collisions to abstract from global cache implementation details.

Analysis

First, we observed particularities of PDB in terms of preallocated constants. Second, even under optimal conditions hash collisions became visible at 2M cached objects. We suspect this becoming a dominant factor in further experiments. This indicates also that the hash code quality should be an engineering priority in case of a "go" decision for maximal sharing for this kind of data.

The naive *method 1* of comparing mean memory is not able to show the effect of maximal sharing due to GC noise. In contrast, our alternative *method 2* shows accurate results that matched our model.

We may confirm **Q-Accurate** for this case: MASHO precisely analyzes potential savings and losses, our second method of memory comparison works, and also `equals`-calls are predicted accurately.

6.7 Evaluation: Case Study

In this section we report on our experience with predicting the effect of maximal sharing in the context of PDB being embedded into Rascal. We will evaluate the following benchmarks:

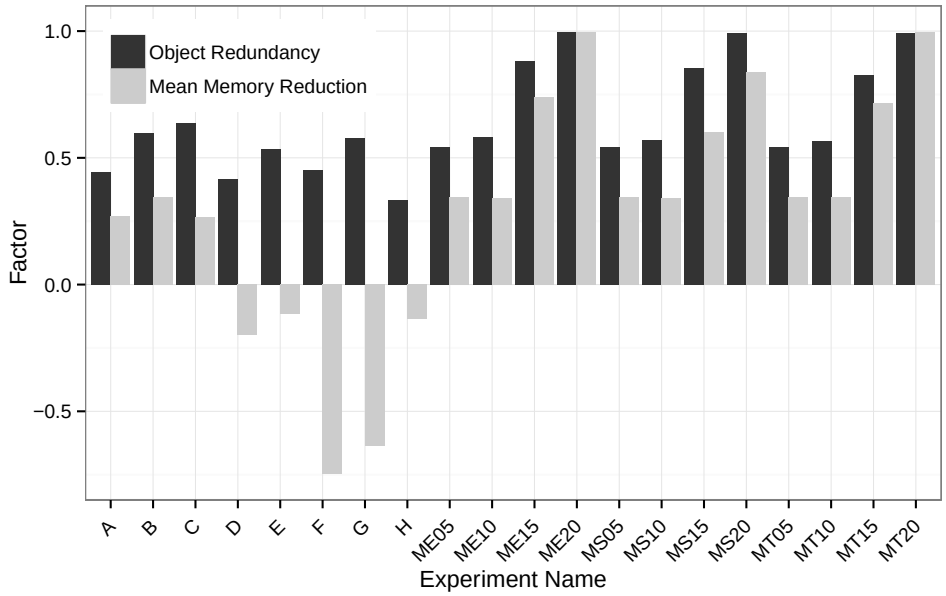


Figure 6.6: Relation of Redundancy and Mean Memory Savings

A: Start the Read-Eval-Print Loop (REPL) of the Rascal language interpreter, and load its prelude module.

B: Start the REPL of the Rascal language interpreter, and generate a parser for an expression grammar.

C: Start the REPL of the Rascal language interpreter, and type check a large module (5–10k lines of code).

D–H: Load serialized call-graphs and calculate the transitive closure for *JHotdraw*, *JWAM16FullAndreas*, *Eclipse202a*, *jdk14v2* and *JDK140AWT*. These benchmarks are supposed to stress the influence of data shape, and the effect of redundancy in algorithmic computation.

M{E,S,T}: Peano arithmetic modulo 17 in three variations, i.e. *expression*, *symbolic*, and *tree*. These are standard benchmarks for term rewriting engines and are previously used to measure the effect of maximal sharing [vdBra+02].

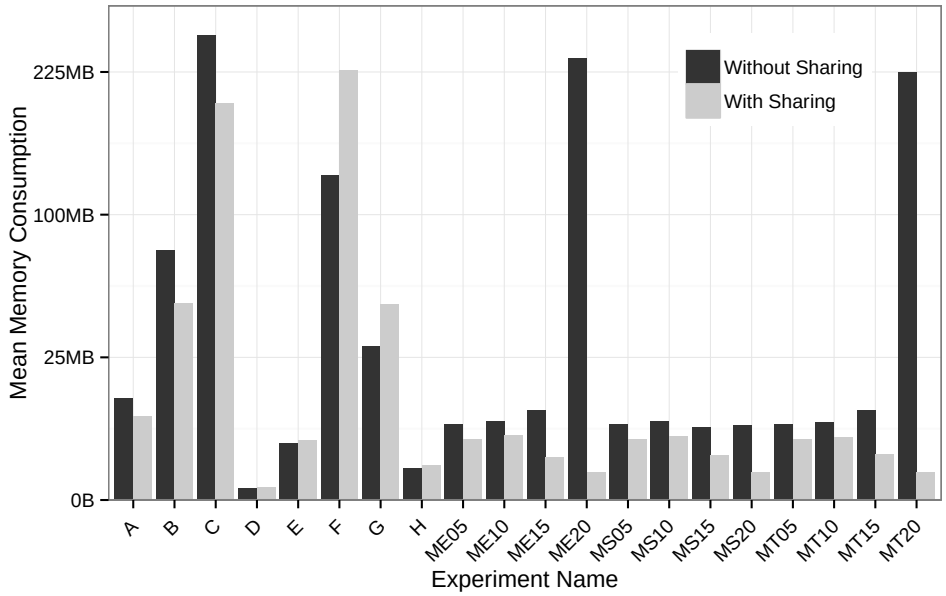


Figure 6.7: Effect of Maximal Sharing on Memory Consumption

Results

First of all, we report that the experimental runs with the maximal sharing aspect of benchmarks B and C timed out after 30 minutes. The cause of the problem, after some manual investigation, was an enormous amount of hashing collisions in the global cache of the shared version. Using MASHO's hashcode logging feature and a Java debugger we found out that the "annotations" feature of PDB was causing trouble. For every annotated value there is a non-annotated value with the same `hashcode`, leading to as many collisions as there are annotated values. In benchmarks B and C there are many parse trees that are annotated with their source code position. To continue our experiments, we then provided an alternative `hashcode` implementations for annotated values, which only the global cache invokes for lookups. Note that altering the problematic `hashcode` method itself is not an option, because it would break the semantic of any program that uses the annotation feature. Applying the fix was necessary for continuing the evaluation, to be able to compare MASHO's models against data from a real maximal sharing implementation. However, the fix was not necessary for a priori performance modeling. We also noticed another `hashcode` related problem —the `hashcode` of a singleton set collides

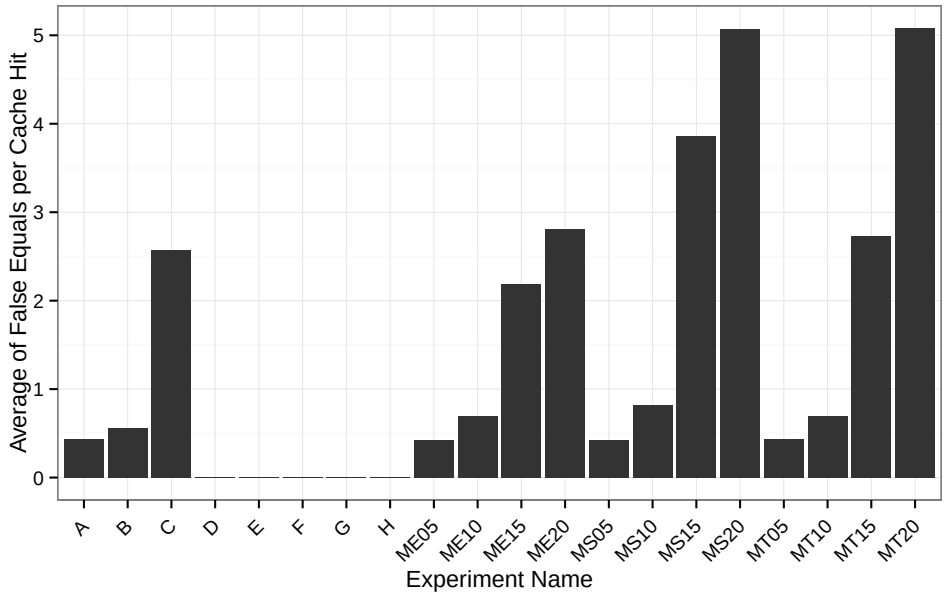


Figure 6.8: Ratio of False Equality Comparisons per Cache Hit

with the `hashcode` of its contained object— and fixed it analogous to the previous problem. Figure 6.12 finally visualizes the results for all benchmarks. We first interpret this data to subsequently answer our evaluation questions. In obtaining the results, ORP and ECP yielded a mean slowdown of 7x (range 2.5–32x).

Object Redundancy and Memory Gains. Figure 6.6 illustrates *object redundancy* in relative numbers, that is how many newly allocated objects yield a hit in the global cache. Over all benchmarks, we can report a mean redundancy of 64%, with a minimum of 33%, and a high of 100% in case of the Peano arithmetic benchmarks. However, the amount of object redundancy does not imply equal gains in *mean memory reduction*. Allover, observed mean memory reductions are below the object redundancy numbers, emphasizing that the size of redundant objects matters and not only their count. In case of the algorithmic transitive closure benchmarks (D–H) we even see a negative net impact on mean memory consumption, albeit 33–58% object redundancy. The loss is attributed to the overhead of the global cache and that redundancy is mostly present in terms of small objects. Figure 6.7 presents another view on the *Mean Memory Reduction* data points from Figure 6.6 by displaying the mean memory usage of the benchmarks before and after applying maximal sharing.

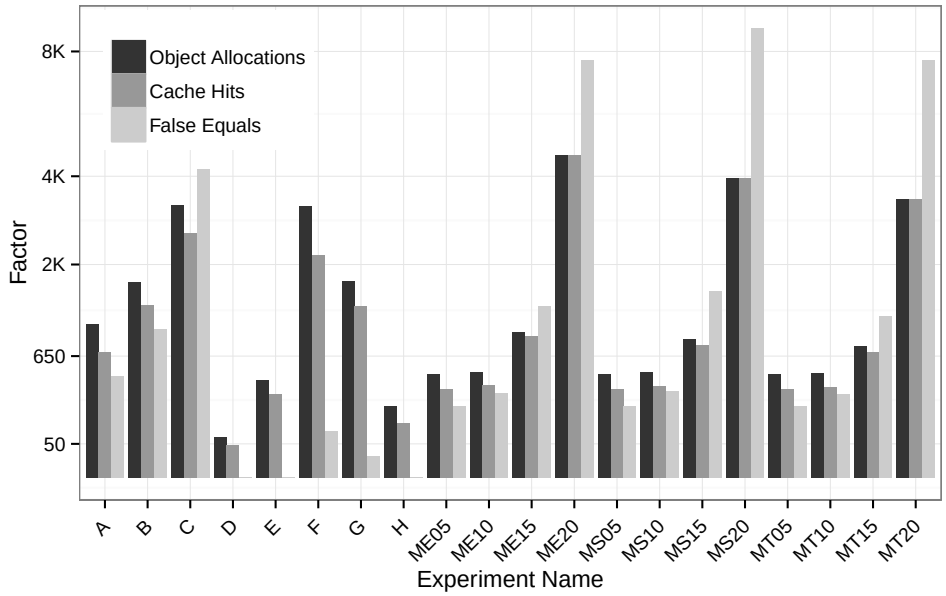


Figure 6.9: Relation of Object Allocation, Cache Hits, and Collisions

Cache Hits and Negative Comparisons due to Chaining and Hash Collisions. In Figure 6.8 we illustrate the number of `false` `equals`-calls that occur on average when performing a global cache lookup that eventually yields a hit. We do not further distinguish and discuss the causes of `false` `equals`-calls, which could be either attributed to implementation details of the global cache (e.g., chaining due to modulo `size` operations), or to hashcode implementations causing collisions. A high ratio should alert a library engineer to systematically explore these possible causes.

Figure 6.9 shows the absolute numbers for object allocations, cache hits, and collisions for all benchmarks. Benchmarks ME20, MS20, and MT20 created a high cache load —causing many negative equality checks— that in the case of ME20 and MT20 led to substantial memory savings (cf. Figure 6.7).

In our data set, on average a global cache hit triggers 1.4 nested reference comparisons. This illustrates how maximal sharing transforms the shape of `equals`-call-graphs: frequent comparisons in the global cache are shallow, and recursive `equals`-calls in the program collapse to one comparison.

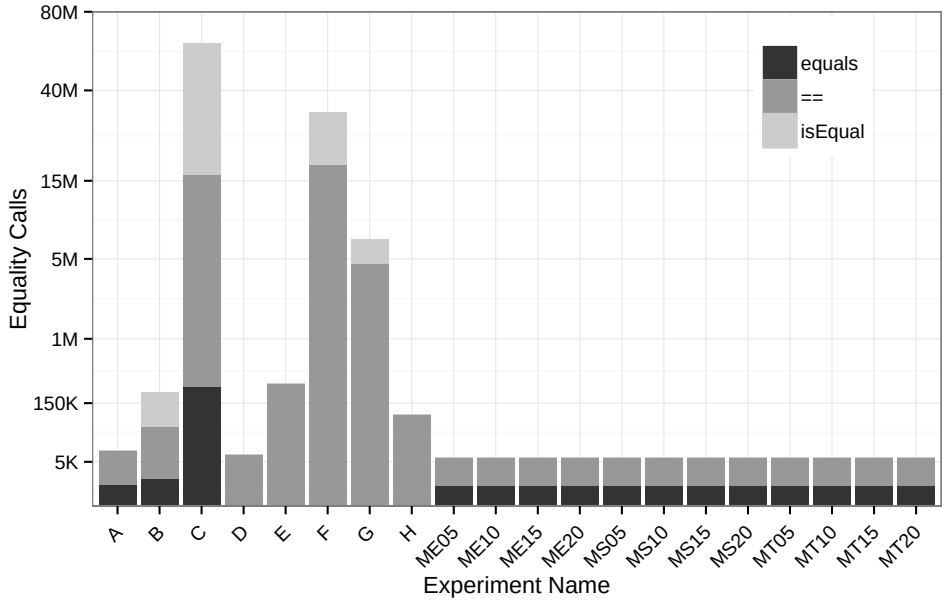


Figure 6.10: Equality Profile without Maximal Sharing

Equality Profile of the Original Library. Figure 6.10 highlights the mixture of equalities encountered. Surprisingly, calls to `equals` with aliased arguments occurred more frequently than calls to `equals` and `isEqual` with distinct arguments. The transitive closure benchmarks D, E and H solely perform reference comparisons. Consequently, the alias-aware analysis of ECP is necessary in our case, otherwise we would have clearly over-approximated savings under maximal sharing. With respect to the recursive call-graph shape of `equals` and `isEqual`, we observed on average 2.7 nested equality calls (other than reference equalities).

Equality Profile with Maximal Sharing. Figure 6.11 shows the equality profile of the experiments with maximal sharing enabled and highlights the changes to Figure 6.10. Absolute numbers of calls decrease, because each recursive `equals`-call is replaced by a single reference comparison. Recursive call-graphs for `isEqual` remain, if two objects are objects are equivalent (`isEqual`) but not strictly equal.

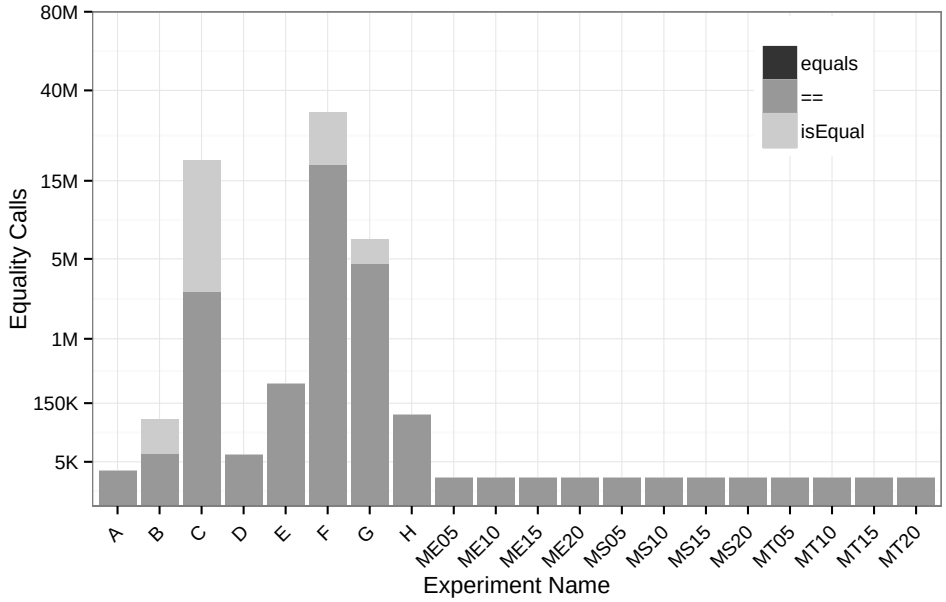


Figure 6.11: Equality Profiles with Maximal Sharing

Figure 6.12: Realistic data: Various memory and equality aspects of the applications under test. Figures 6.7 and 6.9 use a square root scale on the y-axes, and figures 6.10 and 6.11 a double square root scale, to better accommodate the wide range of values.

Benchmarks M{E,S,T} — Peano Arithmetic are designed to bring out best behavior for maximal sharing by generating an enormous amount of redundant terms. The results are shown for different sizes of the problem of symbolically computing 2^n **mod** 17 in Peano arithmetic. The results show that redundancy was accurately predicted for all three benchmark versions. MS exhibited a saving potential up to 86% with increasing input size, the others up to 100%, which is in line with related work [vdBra+02]. However, we do not see significant gains in use of equality. The reason is that our benchmark implementation uses deep pattern matching instead of calls to `equals` and therefore loses the benefit of $O(1)$ reference comparisons.

Analysis

Q-Accurate. None of the experiments showed significant differences between the predicted and the actual memory usage; the mean accuracy of *method 2* was about 99%. For all but one benchmark, calls to `equals` methods were predicted with an accuracy of at least 99%. The only outlier was benchmark B, the parser generator benchmark, that exhibited 19% more calls to `equals`, caused by an corresponding increase in global cache hits. The additional cache hits were caused by equivalent objects that were re-generated at a higher rate than the collection of the weak references from the global cache. In the latter case, we actually under-approximated the potential savings, because the longer living weak references caused an overlap between previously disjoint object families.

We conclude that MASHO accurately models lower bounds for hypothetical heap evolution and calls to `equals` under maximal sharing for all our benchmarks. This means that 256-bit SHA-2 hashes were good enough at least for this (heterogeneous) data, and that the MASHO model is complete.

Q-Actionable. The redundancy data clearly suggests that PDB could benefit from maximal sharing for most benchmarks. However, during profiling we figured out that PDB's annotation feature causes a substantial number of hash-collisions. Furthermore, the effectiveness of maximal sharing diminished under `isEqual` that ignores annotations: many calls to `isEqual` cannot be replaced by reference comparisons. It follows that the library would require severe reengineering before maximal sharing can be applied optimally.

As compared to general memory profilers, which do not consider the specifics and preconditions of maximal sharing, we showed that the GC can hide memory savings of maximal sharing. A memory profiler will not even see substantial savings unless by trial and error global minimum heap bounds are found. Since MASHO ignores the effect of the GC, this confounding effect has become a non-issue.

The information provided by MASHO as compared to our simple maximal sharing aspect is comparable. More importantly, the maximal sharing aspect suffers from arbitrary hash collisions in terms of accuracy (more `equals`-calls will be made as hash buckets become deeper) and speed (the benchmarks will run longer and longer). MASHO provides filtered information, isolating the effect of maximal sharing from the confounding effect of hash collisions.

In additional experiments simulating the semantics of related memory profilers, which check for isomorphic heap graphs [RK14; MO03], we measured that MASHO uncovers up to 14%, and at median 4%, more unique caching opportunities in the aforementioned benchmarks than the related work can provide —due to the additional abstraction facilities.

6.8 Related Work

We position our contribution with respect to memory profiling tools and studies, programming language features, and maximal sharing libraries.

Memory profiling tools and studies. Sewe et al. [Sew+12] investigated the differences in memory behavior between Java and Scala programs. The key findings were that objects in Scala are more likely to be immutable, small, and to have a short lifetime, compared to pure object-oriented Java programs. Ergo, the Scala community may benefit from MASHO.

Dieckmann and Hölzle [DH99] originally published a study about the allocation behavior of six SPECjvm98 Java programs, and compared the results to Smalltalk and ML. The authors obtained allocation data by instrumenting the source code of a virtual machine and built a heap simulator.

Sartor et al. [SHM08] discuss the limits of heap data compression by examining types and sources of memory inefficiencies. Techniques were investigated that work on a wide spectrum of granularity, ranging from object equality to stripping off empty bytes, or compressing fields and arrays. Their analysis approximates saving potentials by analyzing a series of timed heap dumps. The authors observed that deep object equality together with array sharing reduces the size of applications by 14% on average. These results also motivate our research but timed heap dumps do not provide enough detail to assess the impact of maximal sharing accurately.

Resurrector [Xu13] is an object lifetime profiler that supports a tuneable cost settings per allocation site. For frequent calls to allocation sites, Resurrector works more precisely than garbage collector heuristics and can avoid expensive reachability analyses to identify dead objects, as used by like Merlin [Her+06] or Elephant Tracks [RGM13]. These more advanced lifetime profiling techniques are usually implemented inside a Virtual Machine (VM). In contrast, MASHO uses garage collector timestamps as heuristic for object lifetime obtained via JVMTI and BCI and thus works across different JVMs. MASHO predicts with almost perfect accuracy (see Section 6.5), so these more precise and much more expensive techniques are not necessary here.

Bhattacharya et al. [Bha+11] reduce unnecessary allocations of temporary strings or container objects inside loops, by analyzing which objects can be reused after each loop iteration. MASHO reasons over redundancy of a whole program run and therefore also covers these cases, necessarily.

Nguyen and Xu [NX13] detect cacheable objects at allocation sites with variants of data dependence graphs, and memoizable functions at their call sites. Their tool, Cachetor, is implemented inside a VM and targets arbitrary mutable programs and thus leading to a 200x overhead. Redundancy profiling, as implemented by MASHO, in contrast exploits the preconditions of immutable object graphs and can thus operate at lower runtime overheads.

To optimize compilers, Lattner and Adve [LA05] researched a *macroscopic* approach for reasoning over pointer-intense programs, by focusing on how programs use entire logical data structures, rather than individual objects, to then segregate these objects automatically into separate memory pools.

With Object Equality Profiling (OEP), Marinov et al. [MO03] pinpoint groups of equivalent objects that could be replaced by a single representative instance. OEP considers every single object created during a program run. The authors use BCI to track heap activity dynamically. A post-mortem analysis calculates mergeability of objects, by checking isomorphism of labelled graphs. OEP uses an off-line graph partitioning algorithm to process data sets that might exceed main memory size in $\mathcal{O}(n \log n)$ time. One of the key contributions of OEP—that makes it scalable for mutable objects but difficult to apply for modeling maximal sharing— is pre-partitioning heap graphs based on the primitive values of objects as a discriminator. In our context this causes OEP not being able to abstract from implementation details such as arbitrary ordering of elements in arrays, specialized sub-classes, and lazily initialized or cached hashcodes. For libraries based on immutable objects, this can make objects look different while they should be the same. Our experiments showed that MASHO uncovers up to 14%, and at median 4%, more unique sharing opportunities than OEP on the same data would. The focus on immutable objects gives MASHO both the opportunity to abstract and the ability to optimize the necessary high granularity memory profiles.

Rama and Komondoor [RK14] worked on an extension of OEP and introduced a tool, the Object Caching Advisor (OCA), to support introducing hash-consing at the source-code level as a refactoring. The authors reuse a fingerprinting function, introduced by Xu [Xu12], that runs in $O(\textit{size of object graph})$ and yields a runtime overhead ranging from 98–2520x. In contrast, MASHO's fingerprinting, which is based on Merkle trees, operates in $O(\textit{shallow object size})$.

Language support for obviating equals and hashCode. Vaziri et al. [Vaz+07] proposed a declarative, stricter form of object identity called *relation types*. By requiring that the identity of an object never changes during runtime, the authors obviated potential error-prone `equals` and `hashcode` methods. A subset of immutable key fields, referred to as tuple, designates object identity. These tuples match our weak-immutability requirement (see Section 6.2). The authors formalized their programming model and proved that hash-consing preserves semantics and is a safe optimization in their model. Our contribution is nicely orthogonal: Whereas MASHO investigates maximal sharing for libraries and requires that `equals` and `hashcode` are user provided, relation types are meant to be an equality substitute at language level. Vaziri et al. contribute the language supported semantics of weak immutability, which is our a priori assumption.

Scala counters fragile `equals` and `hashcode` implementations with the concept of *case classes*. Scala shows that immutable data types that adhere to structural equality can obviate hand-written `equals` and `hashcode` implementations. The compiler synthesizes their implementation, but since maximal sharing is not always beneficial it does not generate shared implementations. A recast of MASHO to Scala may help finding optimal solutions for libraries that heavily rely on case classes.

The ATerm library. is a prime source of inspiration [vdBra+00; vdBK07; vdBMV05]. Both in C and in Java this is a successful library that employs maximal sharing for representing atomic data-types, lists and trees. Key design considerations of the ATerm library are to specialize garbage collection (in C), and (de)serialization as well based on the condition of maximal sharing and structural equality. In this chapter we use benchmarks from the ATerm experience to evaluate MASHO. The reported use cases of ATerm library (specifically in the term rewriting and model checking context) indicate the possibility of great savings in memory consumption and great increases in performance, but in the general case it is unlikely that maximal sharing is always a good idea.

ShadowVM [Mar+13] is a recent generic Java run-time analysis framework. It separates instrumentation from the client VM and adds asynchronous remote evaluation to increase isolation and coverage. Analyses can be written on a high level of abstraction using an open pointcut model and support bytecode instruction granularity as well. MASHO would be a good usage scenario for ShadowVM, since tracking == instructions needs bytecode instrumentation beyond the capabilities of AspectJ.

6.9 Conclusion

We introduced a new predictive performance modeling tool —named MASHO— for assessing the effects of introducing the maximal sharing optimization strategy into a Java library without changing the library or client code of the library.

MASHO profiles object redundancy and calls to `equals` efficiently using object fingerprints. Under the assumption of *weak immutability*, fingerprinting leads to an accurate model efficiently. MASHO can abstract from accidental implementation details in the current version of a library, such as arbitrary array orderings which also enhances its accuracy.

The experience report focused on the accuracy of the predictions, since fingerprinting and feedback loops with garbage collection heuristics may introduce noise. This showed on a controlled case and on realistic cases that MASHO's predictions are accurate. Predictive performance analysis with MASHO isolates the effect of maximal sharing from noise in the measurements, in contrast to a full blown experiment where confounding effects may be prohibitive for decision making.

Chapter 7

Conclusion

In this dissertation we presented a generative programming perspective for creating efficient immutable collection data structures for standard libraries of programming languages. We analyzed the variability within the domain of collection libraries, and outlined intermediate abstractions for concisely encoding the variability amongst various collection data structures. The key enabler for concisely encoding the variability are novel and efficient data structure designs that improve memory footprints and runtime performance over the Hash-Array Mapped Trie (HAMT) data structure. HAMT is the current state-of-the-art and de-facto standard for implementing efficient unordered hashed collections, such as hash-sets or hash-maps. Form a bird's eye view, we learned the following lessons while doing our research:

Algorithmic Improvements: Algorithmic improvements —that were enabled by novel data structure designs— proved to be the most effective contributions, yielding speedups of several orders of magnitude.

Data Locality: Improving data locality turned out to be a key factor to create industrial strength data structures with good performance, especially under the restrictions of languages with automatic memory management, such as Java or C#, that prohibit fine-grained memory data layouts. Our data structures and algorithms improve data locality by design. Besides that, we consider code specialization of data types essential for reducing memory indirections.

Language and Runtime Support for Immutability: We assess that Java lacks proper programming language abstractions to support the design and implementation of immutable data types. From a user's perspective, the language only offers the `final` keyword for single assignments to variables. However, the Java Virtual Machine (JVM) mostly ignores the final modifier, because reflective code is still able to change the content of final fields.

In the remainder of this chapter we first recapitulate the answers to our research questions (cf. Section 7.1), before outlining remaining challenges and future work (cf. Section 7.2), and concluding (cf. Section 7.3).

7.1 Recapitulation of Research Questions

Research Question (RQ-1): What are the commonalties and differences between different data structures in the domain of collection libraries? Can we generalize over the sub-domain of immutable collections by modeling variability and documenting implementation choices?

RQ-1 is answered by Chapter 2. The chapter describes a small core of intermediate abstractions for immutable collection data structures that covers all variations without exhaustively listing them, and enables excellent runtime performance.

Based on the outcome of our domain analysis and our intermediate abstractions, we designed a product line for immutable collection data structures that relieves library designers from optimizing for the general case, and allows the co-evolution of a collection family (with a potentially large code base) efficiently.

So far related work, as discussed in this thesis, focused on synthesizing complex data structures by composing basic collection data structures (e.g., array-list, linked-list, hash-map). Unlike our approach, none of these results tackled the generation of basic collection data structures themselves. Due to our focus on describing the variability of those basic data structures, future iterations of data structure code generators could apply an integrated code generation approach, fusing various levels of abstraction across basic collections and complex composites.

Research Question (RQ-2): What are the strength and weaknesses of current state-of-the-art persistent data structures that are based on HAMTs, i.e., hash-sets and hash-maps?

RQ-2 is answered by Chapter 3. We identified that the main strength points of HAMTs are incremental updates (for basic operations such insertion and deletion) and structural operations on the partially-ordered trie structure, e.g., a set union operation that merges trees directly instead of using higher level abstractions.

We identified the following weaknesses of HAMT implementations: first, HAMT implementations require dynamic type checks or polymorphism (to distinguish real payload data from accidental tree encoding) due to an insufficient bitmap encoding; second, traversal algorithms suffer from cache locality issues due to an untyped

mix of heterogeneous data; third, operations such as equality checking are not incremental due to potentially degenerated trie structures that are not canonical.

We addressed the above mentioned weaknesses by proposing a new cache-oblivious memory layout, an explicit bitmap encoding that enables a higher compression ratio, and invariants that enforce a canonical tree structure. We call our improved HAMT variant a Compressed Hash-Array Mapped Prefix-tree (CHAMP). CHAMP increases the overall performance of immutable sets and maps, and outperforms Scala's and Clojure's implementations in terms of memory footprint and runtime efficiency of iteration (1.3–6.7 x) and equality checking (3–25.4 x).

CHAMP constitutes the key contribution of this thesis and is the foundation for efficiently encoding the variability of the domain of unordered immutable collections in a Software Product Line (SPL). CHAMP's explicit bitmap encoding avoids dynamic type checks at runtime and reorders objects for improved data locality by design. CHAMP further enables improved algorithms, effective memory footprint specializations, and generalized data structures that support type-heterogeneous content. Overall, CHAMP facilitates generating an efficient and extensive family of collection data structures.

Research Question (RQ-3): HAMT data structures inherently feature many memory indirections due to their tree-based nature. How can we optimize the memory footprint of such data structures on the JVM?

RQ-3 is answered by Chapters 4 and 5. To effectively optimize the memory footprint of trie data structures, it is sufficient to specialize trie-nodes based on their arity. In each specialization we can then replace the node's dynamic content array by a static number of inlined fields. However, HAMTs feature an exponential amount of typed content configurations (and memory specializations) because a trie's content is dynamically dictated by the hash codes of its elements.

In Chapter 4 we explore a pragmatic approach that applies generative programming techniques to specializing HAMT nodes. We reduced the exponential number of specializations to a small practical subset while still maximizing memory savings, by experimentally analyzing typical node-arity distributions. With this techniques we achieved a median decrease of 55 % in memory footprint for maps and 78 % for sets compared to a non-specialized version, but at the cost of 20–40 % runtime overhead of the lookup operation.

In Chapter 5 we improved the scalability and performance over our previous pragmatic approach. Instead of finding a small number of specializations, the improved approach makes use of full specialization, while at the time avoiding code bloat and polymorphic calls for improved runtime performance.

Next to data structure and algorithmic improvements, memory specialization is an effective optimization strategy. Specialized data structures can significantly lower an application's memory footprint, and improve data locality due to a reduced amount of memory indirections and objects created. Memory specialization techniques become feasible through generative programming and code generators and are invaluable for creating high-performance collection data structures.

Research Question (RQ-4): How can HAMTs be utilized (or extended) to efficiently support the implementation of specialized composite data structures such as multi-maps, avoiding the memory overhead of nesting existing collections into each other?

Research Question (RQ-5): Can we bring the advantages in efficiency of primitive collections to generic collections? E.g., how can we optimize data structures for numeric or primitive data types? Or, how can we mix elements of primitive types and reference types without boxing?

RQ-4 and RQ-5 are answered by Chapter 5. The chapter describes a general framework for Hash-Array Mapped Tries on the JVM which can store type-heterogeneous keys and values: a Heterogeneous Hash-Array Mapped Trie (HHAMT). Among other applications, this allows for highly efficient composite data structures such as memory-optimized multi-maps. The new multi-map design reduced the per key-value storage overhead by a factor of two. Moreover, our type-heterogeneous data structure design allows specializing data structures by content. The resulting data structures can freely mix unboxed primitive types and reference types with a factor four improvement in memory consumption over standard generic collections.

Research Question (RQ-6): Hash-consing is an essential optimization strategy to consider for persistent data structures, which involves cross-cutting and invasive changes to a data structure library code base. How can we evaluate a priori the performance impact of applying hash-consing to a data structure library?

RQ-6 is answered by Chapter 6. We detail a priori performance modeling techniques for the hash-consing optimization strategy. We model the potential performance implications of introducing hash-consing to a library that does not implement it yet. Our accompanying performance modeling tool creates predictive performance models, without requiring developers to implement hash-consing.

While in general it is noticeably hard to predict the effect of optimization strategies without implementing them, this thesis illustrates the rewards of doing so. For the case of hash-consing, we have shown that a cheap predictive model can provide more accurate information than an expensive experiment can, and further that we can uncover optimization opportunities that otherwise would remain hidden. We expect that our research approach of a priori modeling the effect of optimization strategies gets picked up by research communities and ported to further optimization strategies. In the end, such tools enable library designers and implementers to make informed choices about the effectivity of optimization strategies upfront.

7.2 Future Work

In this thesis we are concerned with improving the efficiency of immutable collections by devising new data structure designs, algorithms, and memory specialization strategies. We did push the boundaries of immutable data structure implementations under the current version of the JVM. To go beyond this, it is necessary to investigate the co-design of collections with dedicated programming language and Virtual Machine (VM) features for immutability, to boost data structure performance even further. In the remainder of this section we will discuss ideas for carrying research results from this thesis over to other application domains.

Generic Runtime Optimizations for Persistent Data Structures

This section describes ideas on how advanced memory management techniques could help to further increase the performance of tree-based data structures.

Advanced Garbage Collection Strategies. The HAMT data structure imposes a partially-ordered trie structure on hash codes of otherwise unsorted data elements. However, because of automatic memory management, the JVM does not guarantee any particular ordering of the trie data structure and its content in memory. Because immutable object graphs remain stable, a garbage collector could improve on data locality by copying objects according to a HAMT's partial ordering.

A copying garbage collector could increase the locality between tree nodes, by co-locating the objects of partially-ordered trie structures. Such a co-location would essentially flatten tree structures in memory and approximate the locality of array-based structures. Co-location strategies could be triggered transparently upon garbage collection cycles.

Furthermore, long-living object graphs of immutable collections, which have not been modified for longer periods, could benefit from compression strategies that are applied transparently upon garbage collection cycles. Recursive object inlining that omits tree node indirections, or the use of serialization and compression of

immutable heap graphs, are possible optimization strategies to improve the memory footprint of applications that predominantly rely on immutable data structures.

Mapping Persistent Data Structures onto Region-Based Memory Layouts. Efficient immutable data structures that are descendant from HAMTs fall in the category of *persistent* data structures. Persistency, in this context, designates that multiple data structures may structurally share data that they have in common. Recent region-based memory management techniques [Sta+15] allow grouping of memory allocations based on expected object lifetime or semantic. We are interested to explore if region-based memory allocation could be an enabler for advanced, domain-specific memory management strategies for persistent data structures.

Communicating Hints to a Runtime or Garbage Collector. To fully leverage the potential of the two aforementioned research ideas, a data structure engineer would most likely have to signal his/her intents to a language runtime or garbage collector. We are interested to explore solutions that allow signaling intents to components of programming language runtimes.

Optimizing Language Runtimes with Persistent Data Structures

In contrast to statically typed languages, dynamically typed language implementations often impose a significant run-time and memory overhead [Trao9] due to generic collection data structures that can hold *heterogeneously* typed elements. To improve efficiency, language runtimes already optimize collections for *homogeneously* typed primitive elements [BDT13]. The resulting performance improvements mainly stem from object layouts that specialize for unboxed primitive values and corresponding optimized operations.

In many programming language implementations exist homogeneous language abstractions that are decomposed into heterogeneous VM abstractions for which traditional optimizations [BDT13] are not applicable. E.g., the Rascal programming language supports arbitrary-precision integers. From the viewpoint of a language abstraction, a collection of arbitrary-precision integers is homogenous, however from the viewpoint of a VM the data is heterogeneous. VMs internally store integers in the most efficient representation provided by the host language or platform. Integers that fit into the 32-bit range may be stored as an `int` in a JVM language runtime, whereas bigger numbers require complex object representations.

Chapter 5 showcased HHAMT, an efficient encoding that supports (bounded) heterogenous content. Applying HHAMT to runtimes of dynamic programming languages seems very promising to bridge the conceptual differences between high-level language abstractions and low-level VM implementation details.

7.3 Takeaway

Efficient persistent data structures are an important pilar of the success of functional and hybrid functional/object-oriented programming languages, but also of popular productivity tools such as the GIT version control system, or large scale industrial filesystems such as ZFS.

Our results on the CHAMPs data structure design, including our performance results, have already been independently replicated in the ClojureScript programming language, and are being considered for inclusion in JVM languages such as Clojure and Kotlin, amongst others.

This thesis shows that there is still plenty of room to leap the foundations of efficient persistent data structures forward. We are confident that in the near future, immutable collections will become viable defaults over mutable collections. To say it with the words of Pet Helland [Hel15]: *"Immutability Changes Everything"*.

Abstract

This thesis proposes novel and efficient data structures, suitable for immutable collection libraries, that carefully balance memory footprint and runtime performance of operations, and are aware of constraints and platform co-design challenges on the Java Virtual Machine (JVM). 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.

Today's collection data structures are mostly one-off solutions. 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. Applying generative programming techniques may solve the aforementioned problems, however it requires a minimal core that is expressive enough to cover the commonalities and variability of the domain. The key enablers are data structure encodings that constitute the core of this thesis.

First, we contribute a successor to the state-of-the-art approach of Hash-Array Mapped Tries: the Compressed Hash-Array Mapped Prefix-tree (CHAMP). CHAMP improves the overall performance of immutable sets and maps by increasing cache locality and keeping the data structure canonical. Compared to its predecessor, CHAMP reduces memory footprints by design, and most notably increases runtime efficiency of iteration and equality checking significantly.

Second, we propose the Heterogeneous Hash-Array Mapped Trie (HHAMT), a generic encoding that allows storing type-heterogeneous payloads. We detail how a range of data structure design challenges can be reformulated as optimization problems for storing type-heterogeneous payloads. Amongst other optimizations, HHAMT enables highly efficient multi-maps, and collections that efficiently mix (unboxed) value-types und reference types.

Third, we discuss memory layout specialization approaches that are specific to trie-based data structures. Reducing the memory footprint of frequently used collections improves the scalability of programs that handle larger data sets, but also improves application performance in memory constraint environments.

Based on our theoretical results, we generated *Capsule*, a library of trie-based immutable collections that powers the Rascal programming language. Capsule's data structures carefully balance memory footprint and runtime performance, and are tailored toward JVM specifics. We managed to further reduce the performance gap between immutable and mutable collections, and to even surpass mutable collections when it comes to memory footprints and equality checking. Comparisons to state-of-the-art implementations of immutable collections in Clojure and Scala show that our encodings increase overall performance and versatility, making them sensible default choices.

References

[Alv+09] Vander Alves, Daniel Schneider, Martin Becker, Nelly Bencomo, and Paul Grace. "Comparitive Study of Variability Management in Software Product Lines and Runtime Adaptable Systems". In: *VaMoS '09: Proceedings of Third International Workshop on Variability Modelling of Software-Intensive Systems*. Vol. 29. ICB Research Report. Universität Duisburg-Essen, 2009 (cit. on p. 27).

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986 (cit. on pp. 62, 109).

[Bag00] Phil Bagwell. *Fast And Space Efficient Trie Searches*. Tech. rep. LAMP-REPORT-2000-001. Ecole polytechnique fédérale de Lausanne, 2000 (cit. on pp. 28, 67).

[Bag01] Phil Bagwell. *Ideal Hash Trees*. Tech. rep. LAMP-REPORT-2001-001. Ecole polytechnique fédérale de Lausanne, 2001 (cit. on pp. 15, 28, 31, 36, 40, 41, 55, 68, 72, 76, 84, 100).

[Bag02] Phil Bagwell. *Fast Functional Lists, Hash-Lists, Deques, and Variable Length Arrays*. Tech. rep. LAMP-REPORT-2002-003. Ecole polytechnique fédérale de Lausanne, 2002 (cit. on p. 68).

[Bak93] Henry G. Baker. "Equal Rights for Functional Objects or, the More Things Change, the More They Are the Same". In: *SIGPLAN OOPS Messenger* 4.4 (1993) (cit. on p. 121).

[Bal+07] Emilie Balland, Paul Brauner, Radu Kopetz, Pierre-Etienne Moreau, and Antoine Reilles. "Tom: Piggybacking Rewriting on Java". In: *RTA '07: Proceedings of International Conference on Rewriting Theory and Applications*. LNCS. Springer, 2007 (cit. on pp. 8, 116).

[BDT13] Carl Friedrich Bolz, Lukas Diekmann, and Laurence Tratt. "Storage Strategies for Collections in Dynamically Typed Languages". In: *OOPSLA '13: Proceedings of the ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications*. ACM, 2013 (cit. on pp. 87, 93, 111, 148).

[Ber+09] Michael R. Berthold, Nicolas Cebron, Fabian Dill, Thomas R. Gabriel, Tobias Kötter, Thorsten Meinl, Peter Ohl, Kilian Thiel, and Bernd Wiswedel. "KNIME - the Konstanz Information Miner: Version 2.0 and Beyond". In: *SIGKDD Explorations Newsletter* 11.1 (2009) (cit. on p. 112).

[Bha+11] Suparna Bhattacharya, Mangala Gowri Nanda, K. Gopinath, and Manish Gupta. "Reuse, Recycle to De-bloat Software". In: *ECOOP '11: Proceedings of the 25th European Conference on Object-oriented Programming*. Springer, 2011 (cit. on p. 140).

[Bib+15] Aggelos Biboudis, Nick Palladinos, George Fourtounis, and Yannis Smaragdakis. "Streams a la carte: Extensible Pipelines with Object Algebras". In: *ECOOP '15: Proceedings of the 29th European conference on Object-Oriented Programming*. LIPIcs. Schloss Dagstuhl, 2015 (cit. on pp. 36, 84).

[Big98] Ted J. Biggerstaff. "A Perspective of Generative Reuse". In: *Annals of Software Engineering* 5.1 (1998) (cit. on pp. 7, 18, 27, 68, 73).

[Bir77] Richard S. Bird. "Two Dimensional Pattern Matching". In: *Information Processing Letters* 6.5 (1977) (cit. on p. 28).

[BJM13] Thomas Braibant, Jacques-Henri Jourdan, and David Monniaux. "Implementing Hash-consed Structures in Coq". In: *ITP '13: Proceedings of the 4th International Conference on Interactive Theorem Proving*. LNCS. Springer, 2013 (cit. on pp. 8, 116).

[Blo08] Joshua Bloch. *Effective Java, Second Edition*. Addison-Wesley, 2008 (cit. on pp. 8, 88, 89).

[Bor+98] Peter Borovanský, Claude Kirchner, Hélène Kirchner, Pierre-Etienne Moreau, and Christophe Ringeissen. "An Overview of ELAN". In: *WRLA '98: International Workshop on Rewriting Logic and its Applications*. Ed. by Claude Kirchner and Hélène Kirchner. Vol. 15. ENTCS. Elsevier, 1998 (cit. on pp. 8, 116).

[BR11] Phil Bagwell and Tiark Rompf. *RRB-Trees: Efficient Immutable Vectors*. Tech. rep. EPFL-REPORT-169879. Ecole polytechnique fédérale de Lausanne, 2011 (cit. on pp. 15, 29, 43, 68).

[Bru+10] Hugo Bruneliere, Jordi Cabot, Frédéric Jouault, and Frédéric Madiot. "MoDisco: A Generic and Extensible Framework for Model Driven Reverse Engineering". In: *ASE '10: Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*. ACM, 2010 (cit. on p. 112).

[CD08] Pascal Cuoq and Damien Doligez. "Hashconsing in an Incrementally Garbage-collected System: A Story of Weak Pointers and Hashconsing in OCaml 3.10.2". In: *ML '08: Proceedings of the 2008 ACM SIGPLAN Workshop on ML*. ACM, 2008 (cit. on p. 8).

[CE00] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. ACM Press, 2000 (cit. on pp. 7, 18, 27, 68, 73).

[CHK06] Keith D. Cooper, Timothy J. Harvey, and Ken Kennedy. *A Simple, Fast Dominance Algorithm*. Tech. rep. TR-06-33870. Rice University, 2006 (cit. on p. 62).

[Cla+01] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and José F. Quesada. "Maude: Specification and Programming in Rewriting Logic". In: *Theoretical Computer Science* (2001) (cit. on pp. 8, 116).

[CN01] Paul Clements and Linda Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2001 (cit. on p. 27).

[DH99] Sylvia Dieckmann and Urs Hölzle. "A study of the Allocation Behavior of the SPECjvm98 Java Benchmarks". In: *ECOOP '99: Proceedings of the 13th European Conference on Object-oriented Programming*. Springer, 1999 (cit. on p. 140).

[dlBri59] Rene de la Briandais. "File Searching Using Variable Length Keys". In: *IRE-AIEE-ACM '59 (Western): Papers Presented at the March 3-5, 1959, Western Joint Computer Conference*. ACM, 1959 (cit. on pp. 28, 38, 67, 72, 85).

[Dri+86] James R. Driscoll, Neil Sarnak, Daniel D. Sleator, and Robert E. Tarjan. "Making Data Structures Persistent". In: *STOC '86: Proceedings of the Eighteenth Annual ACM Symposium on Theory of Computing*. ACM, 1986 (cit. on pp. 10, 11, 28, 68, 72).

[Ebe+07] Jürgen Ebert, Daniel Bildhauer, Hannes Schwarz, and Volker Riediger. "Using Difference Information to Reuse Software Cases". In: *Softwaretechnik-Trends* (2007) (cit. on p. 38).

[Fre60] Edward Fredkin. "Trie Memory". In: *Communications of the ACM* 3.9 (Sept. 1960) (cit. on pp. 28, 38, 67, 72, 85).

[Got74] Eiichi Goto. *Monocopy and Associative Algorithms in Extended Lisp. University of Toyko.* Tech. rep. 1974 (cit. on pp. 10, 13, 116).

[GS12] Joseph Gil and Yuval Shimron. "Smaller Footprint for Java Collections". In: *ECOOP '12: Proceedings of the 26th European conference on Object-Oriented Programming*. Springer, 2012 (cit. on pp. 52, 69, 77, 111).

[Hal+08] Svein Hallsteinsen, Mike Hinchey, Sooyong Park, and Klaus Schmid. "Dynamic Software Product Lines". In: *Computer* 41.4 (2008) (cit. on p. 27).

[Haw+11] Peter Hawkins, Alex Aiken, Kathleen Fisher, Martin Rinard, and Mooly Sagiv. "Data Representation Synthesis". In: *PLDI '11: Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 2011 (cit. on p. 28).

[Haw+12] Peter Hawkins, Alex Aiken, Kathleen Fisher, Martin Rinard, and Mooly Sagiv. "Concurrent Data Representation Synthesis". In: *PLDI '12: Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 2012 (cit. on p. 28).

[Hel15] Pat Helland. "Immutability Changes Everything". In: *Communications of the ACM* 59.1 (2015) (cit. on pp. 26, 149).

[Her+06] Matthew Hertz, Stephen M. Blackburn, J. Eliot B. Moss, Kathryn S. McKinley, and Darko Stefanović. "Generating Object Lifetime Traces with Merlin". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 28.3 (2006) (cit. on p. 140).

[HK14] Mark Hills and Paul Klint. "PHP AiR: Analyzing PHP systems with Rascal". In: *CSMR-WCRE '14: Proceedings of IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering*. IEEE, 2014 (cit. on pp. 63, 110).

[HO80] Gerard Huet and Derek C. Oppen. *Equations and Rewrite Rules: A Survey*. Tech. rep. CS-TR-80-785. Stanford University, Department of Computer Science, 1980 (cit. on p. 9).

[HVdM06] Elnar Hajiyev, Mathieu Verbaere, and Oege de Moor. "CodeQuest: Scalable Source Code Queries with Datalog". In: *ECOOP '06: Proceedings of the 20th European Conference on Object-Oriented Programming*. Springer, 2006 (cit. on pp. 8, 38).

[IV02] Atsushi Igarashi and Mirko Viroli. "On Variance-Based Subtyping for Parametric Types". In: *ECOOP '02: Proceedings of the 16th European conference on Object-Oriented Programming*. Springer, 2002 (cit. on pp. 36, 84).

[Knu79] Donald Knuth. "Structured Programming with Goto Statements". In: *Classics in Software Engineering*. Ed. by Edward Nash Yourdon. Yourdon Press, 1979 (cit. on p. 116).

[KS01] Andrew Kennedy and Don Syme. "Design and Implementation of Generics for the .NET Common Language Runtime". In: *PLDI '01: Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*. ACM, 2001 (cit. on p. 5).

[KvdSV09] Paul Klint, Tijs van der Storm, and Jurgen Vinju. "Rascal: A Domain Specific Language for Source Code Analysis and Manipulation". In: *SCAM '09: Proceedings of the 9th IEEE International Working Conference on Source Code Analysis and Manipulation*. IEEE, 2009 (cit. on pp. 8, 31, 38, 112, 116, 126).

[LA05] Chris Lattner and Vikram Adve. "Automatic Pool Allocation: Improving Performance by Controlling Data Structure Layout in the Heap". In: *ACM Sigplan Notices* 40.6 (2005) (cit. on p. 141).

[LFN02] Richard E. Ladner, Ray Fortna, and Bao-Hoang Nguyen. "A Comparison of Cache Aware and Cache Oblivious Static Search Trees Using Program Instrumentation". In: *Experimental Algorithmics*. Springer, 2002 (cit. on pp. 36, 69).

[LTE16] Calvin Loncaric, Emina Torlak, and Michael D. Ernst. "Fast Synthesis of Fast Collections". In: *PLDI '16: Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 2016 (cit. on p. 28).

[LW94] Barbara H. Liskov and Jeannette M. Wing. "A Behavioral Notion of Subtyping". In: *Transactions on Programming Languages and Systems (TOPLAS)* 16.6 (1994) (cit. on p. 4).

[LW99] Barbara H. Liskov and Jeannette M. Wing. *Behavioral Subtyping Using Invariants and Constraints*. Tech. rep. CMU-CS-99-156. Carnegie Mellon University, 1999 (cit. on p. 4).

[Mar+13] Lukáš Marek, Stephen Kell, Yudi Zheng, Lubomír Bulej, Walter Binder, Petr Tůma, Danilo Ansaloni, Aibek Sarimbekov, and Andreas Sewe. "ShadowVM: robust and comprehensive dynamic program analysis for the Java platform". In: *GPCE '13: Proceedings of the 12th International Conference on Generative Programming: Concepts & Experiences*. ACM, 2013 (cit. on p. 142).

[McC60] John McCarthy. "Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I". In: *Communications of the ACM* 3.4 (Apr. 1960) (cit. on pp. 11, 13).

[McI68] Doug McIlroy. "Mass-Produced Software Components". In: *Proceedings of NATO Software Engineering Conference*. Ed. by P. Naur and B. Randell. 1968 (cit. on pp. 7, 18, 27, 68, 73).

[Mic68] Donald Michie. "Memo Functions and Machine Learning". In: *Nature* 218.5136 (1968) (cit. on p. 13).

[MO03] Darko Marinov and Robert O'Callahan. "Object Equality Profiling". In: *OOPSLA '03: Proceedings of the ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications*. ACM, 2003 (cit. on pp. 121, 139, 141).

[Mos04] Peter D. Mosses. *CASL Reference Manual, The Complete Documentation of the Common Algebraic Specification Language*. Vol. 2960. LNCS. Springer, 2004 (cit. on pp. 8, 116).

[MS07] Nick Mitchell and Gary Sevitsky. "The Causes of Bloat, the Limits of Health". In: *OOPSLA '07: Proceedings of the ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications*. ACM, 2007 (cit. on pp. 58, 107).

[Nor91] Peter Norvig. "Techniques for Automatic Memoization with Applications to Context-free Parsing". In: *Computational Linguistics* 17.1 (Mar. 1991) (cit. on p. 13).

[NW06] Maurice Naftalin and Philip Wadler. *Java Generics and Collections*. O'Reilly, 2006 (cit. on p. 3).

[NX13] Khanh Nguyen and Guoqing Xu. "Cachetor: Detecting Cacheable Data to Remove Bloat". In: *ESEC/FSE '13: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ACM, 2013 (cit. on p. 140).

[Oka99] Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, June 1999 (cit. on pp. 8, 10, 11, 28, 40, 68, 72, 116).

[OM09] Martin Odersky and Adriaan Moors. "Fighting bit Rot with Types (Experience Report: Scala Collections)". In: *FSTTCS '09: Annual Conference on Foundations of Software Technology and Theoretical Computer Science*. LIPIcs. Schloss Dagstuhl, 2009 (cit. on p. 5).

[ON07] Robert Olsson and Stefan Nilsson. "TRASH A dynamic LC-trie and hash data structure". In: *HPSR '07: Workshop on High Performance Switching and Routing*. IEEE, 2007 (cit. on p. 67).

[OW97] Martin Odersky and Philip Wadler. "Pizza into Java: Translating Theory into Practice". In: *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, 1997 (cit. on pp. 2, 5).

[PB04] Vreda Pieterse and Paul E. Black, eds. *"data structure"*. Dictionary of Algorithms and Data Structures. Dec. 2004. URL: http://www.nist.gov/dads/HTML/dataStructure.html (cit. on p. 1).

[Pro+12] Aleksandar Prokopec, Nathan Grasso Bronson, Phil Bagwell, and Martin Odersky. "Concurrent Tries with Efficient Non-blocking Snapshots". In: *PPoPP '12: Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*. ACM, 2012 (cit. on pp. 15, 29, 68).

[Rad08] Peter Rademaker. "Binary Relational Querying for Structural Source Code Analysis". MA thesis. Universiteit Utrecht, 2008 (cit. on p. 38).

[Ram+95] I. V. Ramakrishnan, Prasad Rao, Konstantinos Sagonas, Terrance Swift, and David S. Warren. "Efficient Tabling Mechanisms for Logic Programs". In: *ICLP '95: Proceedings of the 12th International Conference on Logic Programming*. Elsevier, 1995 (cit. on pp. 8, 38).

[RGM13] Nathan P. Ricci, Samuel Z. Guyer, and J. Eliot B. Moss. "Elephant Tracks: Portable Production of Complete and Precise GC Traces". In: *ISMM '13: Proceedings of the 2013 International Symposium on Memory Management*. ACM, 2013 (cit. on p. 140).

[RK14] Girish Maskeri Rama and Raghavan Komondoor. "A Dynamic Analysis to Support Object-sharing Code Refactorings". In: *ASE '14: Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*. ACM, 2014 (cit. on pp. 139, 141).

[SB97] Yannis Smaragdakis and Don Batory. "DiSTiL: A Transformation Library for Data Structures". In: *DSL'97: Proceedings of the Conference on Domain-Specific Languages*. USENIX Association, 1997 (cit. on p. 28).

[Sew+12] Andreas Sewe, Mira Mezini, Aibek Sarimbekov, Danilo Ansaloni, Walter Binder, Nathan Ricci, and Samuel Z. Guyer. "new Scala() instance of Java: A Comparison of the Memory Behaviour of Java and Scala Programs". In: *ISMM '12: Proceedings of the 2012 International Symposium on Memory Management*. ACM, 2012 (cit. on p. 140).

[SHM08] Jennifer B. Sartor, Martin Hirzel, and Kathryn S. McKinley. "No Bit Left Behind: The Limits of Heap Data Compression". In: *ISMM '08: Proceedings of the 7th International Symposium on Memory Management*. ACM, 2008 (cit. on pp. 121, 140).

[SSS79] Edmond Schonberg, Jacob T. Schwartz, and Micha Sharir. "Automatic Data Structure Selection in SETL". In: *POPL '79: Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. ACM, 1979 (cit. on p. 28).

[ST86] Neil Sarnak and Robert E. Tarjan. "Planar Point Location Using Persistent Search Trees". In: *Communications of the ACM* 29.7 (1986) (cit. on p. 40).

[ST95] Nir Shavit and Dan Touitou. *Software Transactional Memory*. ACM Press, 1995 (cit. on p. 68).

[Sta+15] Codruţ Stancu, Christian Wimmer, Stefan Brunthaler, Per Larsen, and Michael Franz. "Safe and Efficient Hybrid Memory Management for Java". In: *ISMM '15: Proceedings of the 2015 International Symposium on Memory Management*. ACM, 2015 (cit. on pp. 112, 148).

[Sta14] Lukas Stadler. "Partial Escape Analysis and Scalar Replacement for Java". PhD thesis. Johannes Kepler University Linz, 2014 (cit. on pp. 94, 112).

[Ste16] Michael J. Steindorfer. *Towards a Feature Model of Trie-Based Collections*. 2016. DOI: 10.5281/zenodo.59739. URL: https://doi.org/10.5281/zenodo.59739 (cit. on p. 29).

[Stu+15] Nicolas Stucki, Tiark Rompf, Vlad Ureche, and Phil Bagwell. "RRB Vector: A Practical General Purpose Immutable Sequence". In: *ICFP '15: Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*. ACM, 2015 (cit. on pp. 15, 29, 68).

[SV14] Michael J. Steindorfer and Jurgen J. Vinju. "Code Specialization for Memory Efficient Hash Tries (Short Paper)". In: *GPCE '14: Proceedings of the International Conference on Generative Programming: Concepts and Experiences*. ACM, 2014 (cit. on pp. 32, 54, 68, 84, 95, 96, 99, 101, 111).

[SV15] Michael J. Steindorfer and Jurgen J. Vinju. "Optimizing Hash-array Mapped Tries for Fast and Lean Immutable JVM Collections". In: *OOPSLA '15: Proceedings of the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages and Applications*. ACM, 2015 (cit. on pp. 29, 31–33, 84, 87, 94, 98, 100, 105, 106, 109, 110).

[SV16a] Michael J. Steindorfer and Jurgen 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]` (cit. on pp. 31–33).

[SV16b] Michael J. Steindorfer and Jurgen J. Vinju. "Performance Modeling of Maximal Sharing". In: *ICPE '16: Proceedings of the 7th ACM/SPEC International Conference on Performance Engineering*. ACM, 2016 (cit. on p. 26).

[SV16c] Michael J. Steindorfer and Jurgen J. Vinju. "Towards a Software Product Line of Trie-based Collections". In: *GPCE '16: Proceedings of the 2016 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*. ACM, 2016.

[SVY09] Ohad Shacham, Martin Vechev, and Eran Yahav. "Chameleon: Adaptive Selection of Collections". In: *PLDI '09: Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 2009 (cit. on p. 28).

[Tra09] Laurence Tratt. "Dynamically Typed Languages". In: *Advances in Computers*. Elsevier, 2009 (cit. on pp. 111, 148).

[UTO13] Vlad Ureche, Cristian Talau, and Martin Odersky. "Miniboxing: Improving the Speed to Code Size Tradeoff in Parametric Polymorphism Translations". In: *OOPSLA '13: Proceedings of the ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications*. ACM, 2013 (cit. on pp. 28, 68, 95, 112).

[Vaz+07] Mandana Vaziri, Frank Tip, Stephen Fink, and Julian Dolby. "Declarative object identity using relation types". In: *ECOOP '07: Proceedings of the 21th European Conference on Object-oriented Programming*. Springer, 2007 (cit. on pp. 8, 13, 116, 141).

[vdBK07] Mark G. J. van den Brand and Paul Klint. "ATerms for Manipulation and Exchange of Structured Data: It's All About Sharing". In: *Information and Software Technology* 49.1 (2007) (cit. on pp. 13, 117, 142).

[vdBMV05] Mark G. J. van den Brand, Pierre-Etienne Moreau, and Jurgen Vinju. "A Generator of Efficient Strongly Typed Abstract Syntax Trees in Java". In: *IEE Proceedings - Software Engineering* 152.2 (2005) (cit. on pp. 117, 142).

[vdBra+00] Mark G. J. van den Brand, Hayco A. de Jong, Paul Klint, and Pieter A. Olivier. "Efficient Annotated Terms". In: *Software: Practice and Experience* 30.3 (2000) (cit. on pp. 117, 125, 142).

[vdBra+02] Mark G. J. van den Brand, Jan Heering, Paul Klint, and Pieter A. Olivier. "Compiling Language Definitions: The ASF+SDF Compiler". In: *Transactions on Programming Languages and Systems (TOPLAS)* 24.4 (2002) (cit. on pp. 8, 116, 117, 133, 138).

[vDK02] Arie van Deursen and Paul Klint. "Domain-Specific Language Design Requires Feature Descriptions". In: *Journal of Computing and Information Technology* 10.1 (2002) (cit. on p. 29).

[Vis04] Eelco Visser. "Program Transformation with Stratego/XT: Rules, Strategies, Tools, and Systems in StrategoXT-0.9". In: *Domain-Specific Program Generation*. Ed. by C. Lengauer et al. Vol. 3016. LNCS. Springer, 2004 (cit. on pp. 8, 116).

[VN14] Luke VanderHart and Ryan Neufeld. *Clojure Cookbook: Recipes for Functional Programming*. O'Reilly, 2014 (cit. on p. 103).

[Wim08] Christian Wimmer. "Automatic Object Inlining in a Java Virtual Machine". PhD thesis. Johannes Kepler University Linz, 2008 (cit. on p. 112).

[Wöß+14] Andreas Wöß, Christian Wirth, Daniele Bonetta, Chris Seaton, Christian Humer, and Hanspeter Mössenböck. "An Object Storage Model for the Truffle Language Implementation Framework". In: *PPPJ '14: Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java platform*. ACM, 2014 (cit. on p. 53).

[Xu12] Guoqing Xu. "Finding Reusable Data Structures". In: *OOPSLA '12: Proceedings of the ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications*. ACM, 2012 (cit. on p. 141).

[Xu13] Guoqing Xu. "Resurrector: A Tunable Object Lifetime Profiling Technique for Optimizing Real-world Programs". In: *OOPSLA '13: Proceedings of the ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications*. ACM, 2013 (cit. on p. 140).