DE GRUYTER OPEN

JACSM 2014, Vol. 6, No. 1, pp. 27 - 42

DOI 10.2478/jacsm-2014-0002

PERSISTENT COLLECTIONS WITH CUSTOMIZABLE EQUIVALENCE AND IDENTITY SEMANTICS

Konrad Grzanek

IT Institute, University of Social Sciences, Łódź, Poland
kgrzanek@spoleczna.pl, kongra@gmail.com

Abstract

Providing a comprehensive set of mechanisms solving the problem of controlling equivalence and identity requires implementing the functionality for non-sequential containers instrumented with the enriched semantics. Functional programming languages, like Clojure, typically miss the functionality by default. The article presents the design considerations, concepts and implementation details of generalized sets and maps aware of the customizable equivalence and identity together with some usage examples.

Key words: Equivalence testing, semantics, identity, functional programming, Clojure, persistent collections

1 Introduction

Testing object for their equivalence as well as creating criteria of establishing their identity is one of the most important tasks to be realized during mature software implementation process. A developer must consider many factors here, like the typing system of the programming language, either static or dynamic, based on the structural equivalence or using tags. The number and the extent of decisions to be made when designing the identity and equivalence related algorithms is so large and wide that the attempts to make a kind of their uniformization in a single library or *API* requires special approaches. Namely, the mechanisms must be put in a formal shape consisting of a set of extendable equivalence and identity operators. A lack of these mechanisms even in as mature programming languages as *Java* and *Clojure* led to a work on their detailed design and implementation in the latter.

The results of the undertaking were presented in a paper titled *Equivalence in Java and Clojure, Design and Implementation Considerations* [1]. Most of it's content focused on:
- the equivalence and identity abstractions,
- implementation details for Java *primitive types* [3],
- implementation details for *java.lang.Number* [2] derivatives,
- implementation details for some *Clojure* [4], [5] reference types (*clojure.lang.Ratio*, *clojure.lang.BigInt*),
- implementation details for selected *sequential values*: instances of *java.lang.String*, *clojure.lang.ISeq*, *java.util.List*, *kongra.core.Pair.*

The mentioned paper lacked a presentation of concepts as well as implementation elements for other kinds of collections, namely *sets* and *maps* (*associative containers*). The present article may be treated thereafter as a natural continuation of [1] with an aim to present customizable, extendable equivalence and identity realization in the two kinds of containers. One final mark to be made here is we will focus on Clojure persistent *sets* and *maps*, abandoning completely other realizations of *java.util.Set* and *java.util.Map* interfaces [2].

2 Equivalence, Identity and the Persistent Collections in Clojure

Traditionally the functional programming style is associated with a lack of explicitly changeable state. Objects posses *value semantics*, they are immutable *values* like the notions in mathematics. This property opens ways to discuss formally the properties of programs and even sometimes makes the properties of programs provable. From the engineer's point of view the lack of state makes achieving programs' correctness easier (in more complex cases – possible).

Immutable collections are the core of Clojure data structures [5]. They are called *persistent* there, with the term *persistent* meaning their persistence across the operators working on them. In other words, the collection operators in the language are *non-destructive*, they do not modify their arguments. Another property of the persistent collections is their *structure-sharing*. This is a pretty old concept, sequences with structure sharing were common in various dialects of Lisp (see [7], [8]). Nowadays data structures other than the sequential ones are known to have the property, e.g. *hash-tries* [6]. Clojure has the following kinds of persistent collections:

- sequences, including vectors1
- sets
- maps
- records – tagged objects with a map semantics

As stated earlier, our previous paper [1] presented a realization of customizable equivalence and identity operators for sequences (*clojure.lang.ISeq* derivatives) and lists including vectors (*clojure.lang.PersistentVector* <: *clojure.lang.APersistentVector* <: *java.util.List*). As a consequence of having these mechanisms:

```
> (deep= [1M 2N] [1 2])
true
> (deep-hash [1M 2N])
994
> (deep-hash [1 2])
994
```

while with the default operators we have erroneous:

```
> (= [1M 2N] [1 2])
false
> (hash [1M 2N])
-1806861044
> (hash [1 2])
156247261
```

Unfortunately, with persistent sets and maps, things go wrong:

```
(deep= #{1M 2N} #{1 2})

No implementation of method: :binary-deep= of protocol:
#'kongra.identity/WithDeep= found for class: clojure.lang.PersistentHashSet
  [Thrown class java.lang.IllegalArgumentException]2
```

and similarly:

```
(deep= {1M 2N} {1 2})
No implementation of method: :binary-deep= of protocol:
#'kongra.identity/WithDeep= found for class: clojure.lang.PersistentArrayMap
  [Thrown class java.lang.IllegalArgumentException]2
```

1 In an opposition to other sequences, vector elements may be randomly accessed in constant time using indexes, yet the vectors are not lazily evaluated, like other types of sequences may be.

2 Observable when being run either in a standalone application or in the Clojure *REPL*

The same lack of implementation occurs for *kongra.identity/deep-hash* operator. As expected, the default Clojure equivalence and identity operators, exhibit wrongful behavior:

```
> (= #{1M 2N} #{1 2})
false
> (hash #{1M 2N})
1271160563
> (hash #{1 2})
460223544
```

and

```
> (= {1M 2N} {1 2})
false
> (hash {1M 2N})
820711855
> (hash {1 2})
1952097988
```

A thoughtful reader might ask was this lack of implementation an accident or a deliberate decision. The “deep” operators throw errors after all when called with arguments of unsupported kinds. In fact it was a deliberate decision dictated by the following two considerations:

1. There should be no default and “safe” implementation for objects of arbitrary types. Raising errors allows to find out and eliminate unpredictability in the system eagerly.
2. It is impossible to provide the implementation of the customizable equivalence and identity operators for arbitrary *set* or *map* type.

The second point is particularly interesting. In the case of *sequences* and *lists* (see [1]) the way to achieve the desired functionality was iterating over the elements of the collection and either aggregating their *deep-hash* values into a resulting *deep-hash* for the sequence or testing for the *deep=* (equivalence) of the (deeply) compared sequences. This approach must be extrapolated onto sets and maps as it seems to be the only reasonable implementation strategy; a *deep-hash* or *deep=* for any collection is a derivative of *deep-hash* or *deep=* of it's components. Unfortunately, due to various ways a set or a map may be implemented (*hash codes*, *balanced trees*), there is no easy way to provide the “deep” semantics into these kinds of collections[3]; one can't inject neither *deep-hash* nor *deep=* into their internal workings[4].

All this led to making another undertaking of implementing special flavors of persistent sets and maps characterized by:

– possessing the *deep-hash/deep=* semantics,
– generosity in their implementation,
– ability to derive the actual container implementation by relying transparently on a specific *back-end* collection.

The following sections constitute a detailed description of the design decisions and a presentation of their implementation details.

3 DeepEntry

The major idea behind the realization of our sets and maps was to create wrapper instances that would be the actual representations of the objects to store within the container. Using this approach we could forget about reaching for the storage internals at the same time not loosing the ability to inject the customizable equivalence and identity semantics. We may call the wrapper instances the *proxy* objects (behind the *proxy* design pattern as described in [9]), because they allow to modify the original behavior of the stored objects.

The proxy type is called *DeepEntry*. The instances of this class are immutable POJO (*Plain-Old Java Objects*). The following listing shows the static structure of the class:

```
public class DeepEntry {

  private final Object value;

  private int hash; // Default to 0

  private DeepEntry(Object value) {
    this.value = value;
  }
}
```

3 In an opposition to a simple sequence or list a set or map exhibits a whole variety of “views” of it's data in a form of iterators, entry sets, etc. It is a natural and justifiable expectation to these derivative “views” to also provide the “deep” semantics.

4 Testing a set or a map for containing a specified element is the most problematic functionality, as the test is associated with using *deep-hash* and/or *deep=* instead of default *java.lang.Object* methods *hashCode* and/or *equals* in the core storage for these data structures.

Deep entries simply store the value and they offer caching of hash codes, like other immutable Java classes, e.g. *java.lang.String*. The two overridden methods are the core of the implementation:

```
  @Override
  public int hashCode() {
    if (0 == hash && null != value) {
      hash = (Integer) DeepHash.proxy.invoke(value);
    }
    return hash;
  }

  @Override
  public boolean equals(Object obj) {
    if (this == obj) {
      return true;
    }
    final Object result = DeepEquiv.proxy.invoke(
                    value, DeepEntry.uncast(obj));
    return RT.booleanCast(result);
  }
```

As it can be seen, both the identity as well as the equivalence operators expressed this way use the *deep-hash* and *deep=* procedures, represented here by special Java constants *DeepEquiv.proxy* and *DeepHash.proxy*.

DeepEntry belongs in fact to the implementation internals, it has a private constructor, but still there are the useful operators that allow any object to be *coerced* to a *DeepEntry*:

```
public static DeepEntry cast(Object obj) {
  if (obj != null &&
      obj.getClass() == DeepEntry.class) {
    return (DeepEntry) obj;
  }
  return new DeepEntry(obj);
}
```

and *uncasted* afterward if needed:

```
public static Object uncast(Object obj) {
  if (obj != null &&
      obj.getClass() == DeepEntry.class) {
    return ((DeepEntry) obj).value;
  }
  return obj;
}
```

The latter mechanism is also extended to a form of a Clojure procedure, as this procedure is a part of the implementation (referenced to in the following sections):

```
public static final AFn uncastFn = new AFn() {
  @Override
  public Object invoke(Object arg) {
    return uncast(arg);
  }
};
```

4 DeepSet

A *DeepSet*, the set that uses the customizable equivalence and identity semantics is the first kind of collection to be described. In fact it's internals are relatively simple to present.

The construction of the class mimics the implementations of other *persistent sets* within the standard Clojure library – the class is effectively *non-extendable* (by the presence of only private constructor), it has the *value semantics* and it implements a set of useful interfaces:

```
public class DeepSet extends AFn implements IObj,
             IPersistentSet, Set, IHashEq {

  private final APersistentSet impl;

  private DeepSet(APersistentSet impl) {
    this.impl = impl;
  }

}
```

As it can be seen above, a *DeepSet* instance wraps a persistent set (the class *clojure.lang.APersistentSet* is the persistent set abstraction) called ***impl*** here. All objects stored in the *DeepSet* are being actually put into the persistent set after being coerced to a *DeepEntry*. This is a design pattern common both for *DeepSets* and for their associative counterpart – the *DeepMap* (see the next section). A manifestation of this approach can be seen at the following listing presenting the process of creating the set:

```
public static DeepSet create(APersistentSet impl) {
  return new DeepSet(impl);
}
```

```
public static DeepSet create(APersistentSet impl,
                             Object... elements) {
  if (elements.length == 0) {
   return new DeepSet(impl);
  }

  for (Object object : elements) {
   impl = (APersistentSet) impl.cons(
                           DeepEntry.cast(object));
  }
  return new DeepSet(impl);
}
```

The underlined part of the code is the process of performing the actual storage of an object with simultaneous coercing to *DeepEntry*.

With the presence of this approach, implementing equivalence operators for *DeepSet* is trivial; it boils down to calling proper equivalence operators on the underlying ***impls***.

```
@Override
public boolean equiv(Object obj) {
  if (this == obj) {
    return true;
  }
  if (!(obj instanceof DeepSet)) {
    return false;
  }
  DeepSet other = (DeepSet) obj;
  return this.impl.equals(other.impl);
}

@Override
public boolean equals(Object obj) {
  return this.equiv(obj);
}
```

The same can be told about computing the *hash code* using the customizable mechanisms – the work is done in the ***impl*** with the presence of *DeepEntries* and their semantics:

```
@Override
public int hashCode() {
  return impl.hashCode();
}
```

```
@Override
public int hasheq() {
  return hashCode();
}
```

On the Clojure side there is a collection of mechanisms for creating DeepSet instances:

```
(defn deep-hash-set
  ([]
     (DeepSet/create (hash-set)))

  ([& keys]
     (DeepSet/create ^APersistentSet (hash-set) keys)))

  and

(defn deep-sorted-set
  [& keys]
  (DeepSet/create ^APersistentSet
        (sorted-set-by (deep-comparator))
        keys))

(defn deep-sorted-set-by
  [pred & keys]
  ((DeepSet/create ^APersistentSet
        (sorted-set-by (deep-comparator pred))
        keys)))
```

The latter two implemented with the help of a special constructor of *DeepEntry* aware comparators:

```
(defn deep-comparator
  ([]
     (deep-comparator compare))

  ([pred]
     (fn [e1 e2]
       (pred (DeepEntry/uncast e1)
             (DeepEntry/uncast e2)))))
```

There are also the procedures for casting collections into *DeepSets*:

```
(defn deep-set?
  [x]
  (instance? DeepSet x))
```

```
(defn ^DeepSet deep-set
  [coll]
  (if (deep-set? coll)
    coll

    (apply deep-hash-set (seq coll))))
```

and a predefined empty *DeepSet*:

```
(def EMPTY-DEEP-SET (deep-hash-set))
```

All the destructive *java.lang.Set* operators are prohibited on *DeepSet*, as it is a persistent, immutable type. This is why:

```
@Override
public boolean add(Object e) {
  throw new UnsupportedOperationException();
}

@Override
public boolean remove(Object o) {
  throw new UnsupportedOperationException();
}

@Override
public boolean addAll(Collection c) {
  throw new UnsupportedOperationException();
}

@Override
public boolean removeAll(Collection c) {
  throw new UnsupportedOperationException();
}

@Override
public boolean retainAll(Collection c) {
  throw new UnsupportedOperationException();
}

@Override
public void clear() {
  throw new UnsupportedOperationException();
}
```

Finally, after extending the proper protocols[5], namely *WithDeepHash* and *WithDeep=* (see [1]), we get:

```
> (in-ns 'kongra.identity)

> (def s1 (deep-hash-set 1M 2N))
> s1
#{1M 2N}

> (def s2 (deep-hash-set 1 2))
> s2
#{1 2}

> (deep-hash s1)
3
> (deep-hash s2)
3
> (deep= s1 s2)
true
```

Additionally, the semantics expands onto the standard Clojure identity and equivalence operators:

```
> (= s1 s2)
true
> (hash s1)
3
> (hash s2)
3
```

5 DeepMap

The implementation of an associative container differs from the *DeepSet* mostly in the fact that here we store both keys and values, both wrapped within the *DeepEntry*. The *DeepMap* also uses ***impl*** – an internal *back-end* storage, but here it is a persistent map (*clojure.lang.APersistentMap* derivative):

```
public class DeepMap extends AFn implements
         IObj, IPersistentMap, Map,
         Serializable, MapEquivalence, IHashEq {

  private final APersistentMap impl;
```

5 Tedious to present here due to a large amount of source code. For more, see the *kongra/identity.clj* compilation unit.

```
  private DeepMap(APersistentMap impl) {
    this.impl = impl;
  }
}
```

The identity (hash) and equivalence operators are trivial, as in the case of the *DeepSet*:

```
@Override
public boolean equiv(Object obj) {
  if (this == obj) {
    return true;
  }
  if (!(obj instanceof DeepMap)) {
    return false;
  }
  DeepMap other = (DeepMap) obj;
  return this.impl.equals(other.impl);
}

@Override
public boolean equals(Object obj) {
  return this.equiv(obj);
}

@Override
public int hashCode() {
  return impl.hashCode();
}

@Override
public int hasheq() {
  return hashCode();
}
```

Creating *DeepMaps* is associated with performing the process of putting the keys and values into the ***impl*** after wrapping them. One can observe this at the following piece of code:

```
public static DeepMap create(APersistentMap impl,
                              Map other) {
  if (null == other) {
    return create(impl);
  }

  for (Object obj : other.entrySet()) {
    Map.Entry entry = (Map.Entry) obj;
    impl =
        (APersistentMap) impl.assoc(
          DeepEntry.cast(entry.getKey()),
          DeepEntry.cast(entry.getValue()));
  }
  return create(impl);
}
```

For the associative containers we also have a collection of operators on the Clojure side, that allow to create the "deep" maps:

```
(defn deep-hash-map
  ([] (DeepMap/create (hash-map)))

  ([& keyvals]
     (DeepMap/create ^APersistentMap (hash-map)
                      keyvals)))

(defn deep-sorted-map
  [& keyvals]
  (DeepMap/create ^APersistentMap (sorted-map-by
                                     (deep-comparator))
                  keyvals))

(defn deep-sorted-map-by
  [pred & keyvals]
  (DeepMap/create ^APersistentMap (sorted-map-by
                        (deep-comparator pred))
                   keyvals))
```

or casting arbitrary maps into their "deep" counterparts:

```
(defn ^DeepMap deep-map
  [m]
  (cond (deep-map? m)
        m

        (instance? clojure.lang.Sorted m)
        (let [c (.comparator ^clojure.lang.Sorted m)]
          (DeepMap/create ^APersistentMap
                    (sorted-map-by (deep-comparator c))
                    ^java.util.Map m))

        (instance? java.util.SortedMap m)
        (let [c (.comparator ^java.util.SortedMap m)]
          (DeepMap/create ^APersistentMap
                    (sorted-map-by (deep-comparator c))
                    ^java.util.Map m))
        :else
        (DeepMap/create ^APersistentMap (hash-map)
                        ^java.util.Map m)))
```

As in the case of the *DeepSet*, *DeepMaps* ensure their non-destructive nature by raising exceptions on an attempt to call destructive operators (belonging to the *java.util.Map* contract). For the sake of simplicity we do not present the related source codes here.

Testing for keys and values containment is as trivial as in the case of the identity and equivalence implementation after wrapping the arguments withind *DeepEntry*:

```
@Override
public boolean containsKey(Object key) {
  return impl.containsKey(DeepEntry.cast(key));
}

@Override
public boolean containsValue(Object value) {
  return impl.containsValue(DeepEntry.cast(value));
}
```

and similarly in the case of retrieving values stored under the given keys:

```
@Override
public Object get(Object key) {
  return DeepEntry.uncast(impl.get(DeepEntry.cast(key)));
}
```

One final interesting implementation aspect is the algorithm for building the derivative collections of *map* entries, keys and values. They are all built around the same pattern – instances of *kongra.utils.decorators.ImmutableDecoratingSet*[6] are used as shown below:

```
@SuppressWarnings("unchecked")
@Override
public Set entrySet() {
  return new ImmutableDecoratingSet<Map.Entry,
                                    Map.Entry>
           (impl.entrySet()) {
    @Override
    protected Map.Entry decorate(Object o) {
      Map.Entry entry = (Map.Entry) o;
```

6 Full presentation of this interesting collection, whose range of possible applications is by no means limited to the functionality presented here, goes beyond the scope of this article.

```
      return new MapEntry(DeepEntry.uncast(
                          entry.getKey()),
          DeepEntry.uncast(entry.getValue()));
    }
  };
}
@SuppressWarnings("unchecked")
@Override
public Set keySet() {
  return new ImmutableDecoratingSet(impl.keySet()) {
    @Override
    protected Object decorate(Object o) {
      return DeepEntry.uncast(o);
    }
  };
}
```

or – in the case of *java.util.Map.values* method implementation – the *kongra.utils.decorators.ImmutableDecoratingCollection*[6] is used instead:

```
@SuppressWarnings("unchecked")
@Override
public Collection values() {
  return new ImmutableDecoratingCollection(
              impl.values()) {
    @Override
    protected Object decorate(Object o) {
      return DeepEntry.uncast(o);
    }
  };
}
```

There are few more elements of the implementation of the *DeepMap* container whose presentation in this paper was postponed for the sake of the overall clarity.

Using the container is straightforward and it leads to the desired behaviors:

```
> (def m1 (deep-hash-map 1M 2N))
> m1
{1M 2N}
> (def m2 (deep-hash-map 1 2))
> m2
{1 2}
> (deep= m1 m2)
true
> (deep-hash m1)
3
```

```
> (deep-hash m2)
3
```

As with *DeepSets*, here we also have:

```
> (= m1 m2)
true
> (hash m1)
3
> (hash m2)
3
```

6 Conclusions

This paper finalizes a series of articles related to the customizable equivalence and identity in Clojure. The presented mechanisms should be used where needed, optionally extended on demand by modifying and/or extending the protocols presented in [1] and referenced to in this article. Establishing the specific performance profiles of the algorithms shown is left to the discretion of their users.

References

1. Grzanek K., 2013, *Identity in Java and Clojure, Design and Implementation Considerations*, Journal of Applied Computer Science Methods, No. 2 Vol. 5 2013
2. Oracle, 2014, *Java™ Platform, Standard Edition 8 API Specification*, http://docs.oracle.com/javase/8/docs/api/
3. Gosling J., Joy B., Steele G., Bracha G., 2005, *The Java™ Language Specification Third Edition*, ISBN 0-321-24678-0, available at the Oracle Technology Network (2014) http://docs.oracle.com/javase/specs/
4. Halloway S., 2009, *Programming Clojure*, ISBN: 978-1-93435-633-3, The Pragmatic Bookshelf
5. Emerick Ch., Carper B., Grand Ch., 2012, *Clojure Programming*, O'Reilly Media Inc., ISBN: 978-1-449-39470-7
6. Bagwell P., 2000, *Ideal Hash Trees (Report)*, Infoscience Department, École Polytechnique Fédérale de Lausanne
7. Touretzky D.S., 1990, *COMMON LISP: A Gentle Introduction to Symbolic Computation*, The Benjamin/Cummings Publishing Company,Inc., ISBN: 0-8053-0492-4
8. Graham P., 1993, *On Lisp - Advanced Techniques for Common Lisp*, Prentice Hall; 1st edition (September 9, 1993), ISBN-10: 0130305529, ISBN-13: 978-0130305527
9. Gamma, et al., E., 1995., *Design Patterns*. Reading, MA: Addison-Wesley Publishing Co, Inc. pp. 175ff. ISBN: 0-201-63361-2