EQUIVALENCE IN JAVA AND CLOJURE, DESIGN AND IMPLEMENTATION CONSIDERATIONS

Konrad Grzanek

IT Institute, University of Social Sciences
9 Sienkiewicza St., 90-113 Łódź, Poland
kgrzanek@spoleczna.pl, kongra@gmail.com

Abstract

Immutability and the functional programming style demand an extensible and generic approach in the domain of semantic and structural equivalence testing. The lack of a library or a framework offering such functionality for Clojure programming language led to some design and implementation efforts that this article undertakes to describe. Incidentally it tries to gather and present a collection of most severe mistakes that may be made by a programmer that attempts to test objects of various kinds for their equivalence, both in Clojure and the underlying Java run-time with it's standard library, showing simple yet usable ways to avoid them.

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

1 Introduction

Growing multitasking programming needs and the popularity of functional programming style brought the notions of immutability and state to the forefront of elements a software engineer must think of when designing and implementing modern software systems. Immutable objects that are commonly associated with mathematical models of the real world make the structural equality a default choice, in the opposition to the explicitly expressed equality, based on an explicit identifier, physical memory location etc., that must be used under an assumption of the always present change. Additionally and in a resulting way, duck typing (see e. g. [1]) is a programming means of abstraction of a growing importance at least in some kinds of systems. This goes in an analogous ways in an opposition to the tag-based typing. Unfortunately, the state of the art in programming languages, even the most advanced ones is not an optimal one when talking about the objects' identity and structural equivalence. The paper gives an overview of these problems and tries to present a generalized solution based on some solid abstractions. Then the important implementation details of an identity framework for Clojure is presented.

2 Reference Types Equivalence Problems

If we assume a concentration on the structural equivalence issues, then the lack of a generalized and extensible solution to the problem of both in Java and Clojure is apparent. These two languages are mentioned here for the following four reasons:

1. Java is a typical, strongly and statically typed programming language [2], very representative for a class of languages used widely nowadays and known as the object-oriented ones. The default identity is the memory location-based one.
2. There are multiple reasons why implementing a non-default identity criteria by overriding the ***java.lang.Object equals*** and ***hashCode*** methods is hard and error-prone [3, 4]. Taking a detailed look at these mechanisms and problems laying there is beyond the scope of this article, but will be presented elsewhere in the future works.
3. Clojure is a modern functional language [5, 6], supporting immutability and using ***Software Transactional Memory*** where the explicit state must be used to achieve a desired functionality. Clojure is strongly typed but in an opposition to Java it lacks static type-checking and uses duck-typing where possible.
4. The two languages both run on top of the JVM, Clojure shares Java libraries and is capable to run an arbitrary Java code, on the other hand embedding Clojure run-time in a Java application is an easy task. One can say these languages are related worlds despite the fundamental stylish and typing differences between them.

Clojure standard library as well as some run-time elements support structural equivalence with respect to collections, in particular. Sequences (vectors, lists), sets and associative collections (maps, records) all exhibit support for deep, structural comparison. Unfortunately, this support is not extensible. Yes, a presence of some interfaces suggests that the mechanisms are capable of being extended, but:

- There are some implementation details that effectively block extending the run-time abstractions with custom classes, written either in Clojure (records, types) or Java (classes). An example of this is introducing a new

composite numeric type, like a Complex number[1]. The new non-atomic numeric type does not fit into Clojure equivalence mechanisms for numbers and there is no way to solve this problem without making significant changes to the core of the language.
- The situation gets disclosed when trying to integrate an existing any computational library into the Clojure based system.
- Even if there were no barriers described above, using the default interface-based abstractions is impossible on already written types (Java classes in particular). Using AOP as described by Kiczales [7] is not an elegant nor easily accessible solution here.

All these problems are easily solvable with use of Clojure ***protocols*** [6], but currently there are no libraries of this kind. This is a very important premise that influenced creating a universal solution described in this article.

3 Equivalence of Numeric Values – Quirks and Corner Cases

Problems described in the previous section expand onto the primitive types, their values as well as their boxed counterparts. To focus our considerations, the general contract for equality and hashing must be provided to the reader. Java Language Specification [2] as well as some other resources [8] say that ***equals*** method implements an equivalence relation. It is:
- Reflexive: For any non-null reference value *x*, *x.equals(x)* must return *true*.
- Symmetric: For any non-null reference values *x* and *y*, *x.equals(y)* must return *true* if and only if *y.equals(x)* returns *true*.
- Transitive: For any non-null reference values *x*, *y*, *z*, if *x.equals(y)* returns *true* and *y.equals(z)* returns *true*, then *x.equals(z)* must return *true*.
- Consistent: For any non-null reference values *x* and *y*, multiple invocations of *x.equals(y)* consistently return *true* or consistently return *false*, provided no information used in equals comparisons on the objects is modified.
- For any non-null reference value *x*, *x.equals(null)* must return *false*.

For the ***hashCode*** method, the following set of constraints applies:
- Whenever it is invoked on the same object more than once during an execution of an application, the hashCode method must consistently return the same integer, provided no information used in equals comparisons on the object is modified. This integer need not remain consistent from one execution of an application to another execution of the same application.

1 Complex numbers are not present in Clojure by default. ANSI Common Lisp ([9]) supports them.

- If two objects are equal according to the equals(Object) method, then calling the hashCode method on each of the two objects must produce the same integer result.
- It is not required that if two objects are unequal according to the equals(Object) method, then calling the hashCode method on each of the two objects must produce distinct integer results. However, the programmer should be aware that producing distinct integer results for unequal objects may improve the performance of hash tables.

Neither the Java primitives[2] nor the derivatives of *java.lang.Number* possess the semantically correct implementations of equivalence mechanisms as a whole. Saying "semantically correct" we mean a correct behavior of the proper methods and operators with respect to *Liskov substitution principle* [10]. Moreover, using some values of these types lead to surprising results, especially the floating-point values representation in Java[3] causes real headaches when attempting to implement solid numeric codes.

The rest of this section is an attempt to present a catalog of semantically incorrect behaviors of numeric values. All examples are given in Clojure, and so we focus on boxed types rather than the primitive ones. We also use *clojure.core/=* and *clojure.core/hash-code* operators instead of calling *equals* and *hashCode* explicitly[4].

There are the most important examples of malfunctioning equivalence in Clojure and Java:

- Erroneous floats equivalence, both in primitive type values and in the boxed ones. Example:

```
> (= (float 1.234) 1.234)
false

> (hash 1.234)
-146307282

> (hash (float 1.234))
1067316150
```

This turns out to be eventually a conversion problem between floats and doubles, because when applying *clojure.core/=* operator the Clojure run-time

2 In the case of primitives we mean the == operator in Java, not the *equals/hashCode* complementary set of methods that apply only for reference types.

3 And all languages with standard IEEE 754 ([11]) floating-point representation.

4 The reader familiar with Clojure should be aware that these operators semantically wrap *equals* and *hashCode*.

performs the auto-promotion of *float* (*java.lang.Float*) value into *double* (*java.lang.Double*). The same happens while executing *clojure.core/hash*.

- Semantically incorrect java.math.BigDecimal behavior:

```
> (= 1.234M 1.2340M)
false

> (hash 1.234M)
38257

> (hash 1.2340M)
382544
```

In the mathematical sense the two literals *1.234M* and *1.2340M* denote the same *java.math.BigDecimal* value. Unfortunately, the representations of the two values differ in scale (see *java.math.BigDecimal* source code, e g. 12]), and as a result the desired equivalence is not reached.

- Erroneous float/double and java.math.BigDecimal equivalence behavior:

```
> (= 1.234 1.234M)
false

> (= (float 1.234) 1.234M)
false

> (hash 1.234M)
38257

> (hash 1.234)
-146307282

> (hash (float 1.234))
1067316150
```

Here we also have a conversion-related problem. Creating a BigDecimal value out of a double one may lead to the following observation:

```
> (java.math.BigDecimal. 1.234)
1.2339999999999999857891452847979962825775146484375M
```

This is hardly an expected result both for a novice and an experienced programmer, unless he does have strong mathematical background in the part of numerical analysis that refers to floating-point representations, or at least is familiar with in this particular Java implementation behaviors.

- *java.math.BigInteger* / *clojure.lang.BigInt* and *java.math.BigDecimal* equivalence relation incompatibility for mathematical integrals:

  ```
  > (= 1N 1M)
  false

  > (hash 1N)
  1

  > (hash 1M)
  31
  ```

  where

  ```
  > (class 1N)
  clojure.lang.BigInt
  ```

  and

  ```
  > (class 1M)
  java.math.BigDecimal
  ```

- Non-equivalence of *clojure.lang.Ratio* [6] and Java floating-point values:

  ```
  > (= 1/4 0.25)
  false

  > (hash 1/4)
  5

  > (hash 0.25)
  1070596096
  ```

  where

  ```
  > (class 1/4)
  clojure.lang.Ratio
  ```

  and

  ```
  > (class 0.25)
  java.lang.Double
  ```

- Lack of equivalence between *clojure.lang.Ratio* and *java.math.BigDecimal* values:

  ```
  > (= 1/4 0.25M)
  false

  > (hash 1/4)
  5

  > (hash 0.25M)
  777
  ```

There are even more problems in the numerical analysis domain that have their source in inconsistent representations, e. g. testing for the equality of two floating-point numbers. For more information about Java's specific numerical quirks one can look into [13].

The author's intent was to create an extensible library with a set of semantic and structural equivalence semantics for Clojure programming language. The library would be a solution to all the mentioned problems and malfunctioning representations both in primitive numeric types with their boxed counterparts and in other reference type values. The rest of presents the design implementation of the most important elements of the library.

4 Common Semantic and Structural Comparison Abstraction

Acceding to the task of designing and implementing the library one of the key decisions to make was choosing the right way of covering a set of existing types with the new abstraction. A non-functional requirement was to stay close to the performance[5] of the original *clojure.core/=* and *clojure.core/hash* operators with the newly implemented versions. This the most important reason why the decision was made to use Clojure ***protocols*** over class-dispatched ***multimethods***, although the former would be more flexible and elegant – testing for equivalence requires in fact *multi-class* dispatch (on two classes in the case of the binary operators).

Deep, semantic and structural equivalence is described by the following protocol:

```
(defprotocol WithDeep=
  (binary-deep=          [this other])
```

5 Regarding the cost of sole procedure call. Due to the semantic richness of the newly designed mechanisms expanding the requirement onto the total performance of the calls would be unrealistic.

```
  (binary-deep=Long       [this other])
  (binary-deep=BigInteger [this other])
  (binary-deep=BigInt     [this other])

  (binary-deep=Double     [this other])
  (binary-deep=BigDecimal [this other])
  (binary-deep=Ratio      [this other])

  (binary-deep=List       [this other])
  (binary-deep=ISeq       [this other])

  (binary-deep=String     [this other])
  (binary-deep=Symbol     [this other])
  (binary-deep=Keyword    [this other])

  (binary-deep=nil        [this]))
```

Other types, the *java.lang.Number* derivatives in particular, may be reduced to the forms described by the methods of this protocol.

The protocol echoes the Visitor design pattern (as described in [14]), that is a common technique for implementing binary dispatch in programming languages where the feature is not present.

Additionally we have a pair of operators that introduce the variadic arity to the library. The implementation of the operators closely mimics the original *clojure.core/=* and *clojure.core/not=* procedures:

```
(defn deep=
  ([x] true)
  ([x y] (binary-deep= x y))
  ([x y & more]
     (if (binary-deep= x y)
       (if (next more)
         (recur y (first more) (next more))
         (binary-deep= y (first more)))
       false)))
(defn deep-not=
  ([x] false)
  ([x y] (not (deep= x y)))
  ([x y & more]
     (not (apply deep= x y more))))
```

Every user of the library is strongly encouraged to use *deep=* and *deep-not=* instead of their binary “colleagues” - the former should be treated only as the slots for injecting new equivalence semantics (when extending the *WithDeep=* protocol).

5 Common Semantic and Structural Hash Code Abstraction

The semantic and structural hash code abstraction is defined as the following simple protocol:

```
(defprotocol WithDeepHash
  (^Integer deep-hash [this]))
```

6 Selected Implementation Details for Numeric Types

Solving the equivalence issues described in section 3 of this paper was based on the design principle of making an optional promotion (either an *upgrade*, *downgrade* or *"horizontal"*) either on the semantic level (algorithmically) or "physically" (by conversion) of the arguments of *deep=* or *deep-hash* according to the following rules (illustrated with the source codes below):

- All Java primitive integrals participating in the sub-typing relation according to the formula *byte* <: *short* <: *int* <: *long*[6] and their boxed counterparts *Byte*, *Short*, *Character*, *Integer*, *Long* are always promoted to *long* / *Long*[7]. Then the promoted value is used to compute the hash code for primitive *long*[8]:

```
(extend-protocol WithDeepHash
  java.lang.Byte
  (deep-hash [this] (Longs/hashCode (.longValue this)))

  java.lang.Short
  (deep-hash [this] (Longs/hashCode (.longValue this)))

  java.lang.Character
  (deep-hash [this] (Longs/hashCode (chlong9 this)))

  java.lang.Integer
  (deep-hash [this] (Longs/hashCode (.longValue this)))

  java.lang.Long
  (deep-hash [this] (Longs/hashCode (.longValue this))))
```

6 Including *char* despite it's own sub-typing path: *char* <: *int* <: *long*

7 Actually, the promotion always takes place on the boxed values, due to the *object-oriented* nature of Clojure protocols. This applies to all mechanisms described in this and the remaining sections.

8 Using utility class *com.google.common.primitives.Longs* from the *Guava* [15] library

9 java.lang.Character is not a Number derivative, so it must be converted into a long beforehand with: `(defn chlong [^Character c] (long (int (.charValue c))))`

- Values of *java.math.BigInteger* and *clojure.lang.BigInt* are treated as *Long* values (being semantically downgraded) iff they fit into the *long* integral range. Otherwise their native *hashCode* implementations are used:

```
(defn BigInteger-in-long?
  [^java.math.BigInteger n]
  (< (long (.bitLength n)) 64))

(extend-protocol WithDeepHash
  java.math.BigInteger
  (deep-hash [this]
    (if (BigInteger-in-long? this)
      (Longs/hashCode (.longValue this))

      (.hashCode this)))

  clojure.lang.BigInt
  (deep-hash [this]
    (if-let [bipart (.bipart[10] this)]
      (.hashCode bipart)

      (Longs/hashCode (.lpart[10] this)))))
```

- Instances of *java.math.BigDecimal* get converted into *java.math.BigInteger* iff they represent an integral value (this may be called a “horizontal” conversion, neither upgrade nor downgrade), or are used as themselves only with stripping trailing zeros to eliminate the scaling incompatibilities:

```
(defn- BigDecimal-deep-hash
  [^BigDecimal d]
  (if (BD/integer?[11] d)
    (deep-hash (.toBigInteger d))

    (.. d stripTrailingZeros hashCode)))

(extend-protocol WithDeepHash
  java.math.BigDecimal
  (deep-hash [this] (BigDecimal-deep-hash this)))
```

10 See [6].

11 The functionality is implemented as a separate custom library whose full description goes beyond the scope of this article, however some of it's details will be presented further in the section. To be required like: `(require 'kongra.bidec :as BD)`.

- Floating-point values are checked against their *infinity* and *NaN* (*not a number*) equivalence. If they are infinite or *NaN* the typical conversion to *long* value is used. Otherwise if they are semantically mathematical integers, they undergo downgrade to *long* value. Finally if neither applies, the *Double* gets promoted to a *java.math.BigDecimal* value:

```
(defn- Double-deep-hash
  [^Number this]
  (let [d (.doubleValue this)]
    (if (or (Double/isNaN d) (Double/isInfinite d))
      (Longs/hashCode (Double/doubleToLongBits d))

      (if (DoubleMath[12]/isMathematicalInteger d)
        (Longs/hashCode (.longValue this))

        (.hashCode (BigDecimal/valueOf[13] d))))))

(extend-protocol WithDeepHash
  java.lang.Float
  (deep-hash [this] (Double-deep-hash (fldouble this)))

  java.lang.Double
  (deep-hash [this] (Double-deep-hash this)))
```

For *Floats* a correct conversion into *Double* values requires an intermediate *String* to take part in the process, in a similar way it was when promoting the *Double* into a *BigDecimal*:

```
(defn ^Double fldouble
  [^Number f]
  (Double/valueOf (Float/toString (.floatValue f))))
```

- Finally, the rational numbers, instances of *clojure.lang.Ratio* undergo a slightly more advanced numerical study to verify whether or not it is possible to convert them into a *java.math.BigDecimal* instances. They are convertible iff they are not *recurring decimals*, i. e. the *prime factorization* of their denominators contains only the numbers 2 and 5, possibly repeating multiple times. When a *Ratio* is not convertible, it's default *hashCode* value is taken as the result:

12 Using utility class *com.google.common.math.DoubleMath* from the *Guava* library.

13 Promoting a *Double* into a *java.math.BigDecimal* requires an intermediate String to take par t in the process. This is the only correct to do this due to the nature of the primitive floating-point representation (sic!). This is an operation of a significant performance cost that must be incurred in the name of the correctness.

```
(defn non-repeating-denominator?
  [n]
  (cond (or (= 1 n) (= -1 n)) true
        (even? n) (recur (/ n 2))
        (= 0 (mod n 5)) (recur (/ n 5))
        :else false))

(extend-protocol WithDeepHash
  clojure.lang.Ratio
  (deep-hash [this]
    (let [denom (.denominator this)]
      (if (non-repeating-denominator? denom)
        (let [num   (BigDecimal. (.numerator this))
              denom (BigDecimal. denom)]

          (BigDecimal-deep-hash (.divide num denom)))
        (.hashCode this)))))
```

The above rules and their realizations in the presented source code referred all to the *deep-hash* procedure. But the same or similar principles influenced the binary (and so the general) *deep*= implementation.

- All primitive integrals and their boxed versions are compared with one another after a promotion to a *Long* type:

```
(defn Long=Long
  [^Number this ^Number other]
  (= (.longValue this) (.longValue other)))
```

For every type either numeric or not the binary equivalence protocol is extended in a way like below for the *Byte* class:

```
(extend-protocol WithDeep=
  java.lang.Byte

  (binary-deep=           [this other] (binary-deep=Long other this))

  (binary-deep=Long       [this other] (Long=Long        other this))
  (binary-deep=BigInteger [this other] (BigInteger=Long  other this))
  (binary-deep=BigInt     [this other] (BigInt=Long      other this))

  (binary-deep=Double     [this other] (Double=Long      other this))
  (binary-deep=BigDecimal [this other] (BD/=             other this))
  (binary-deep=Ratio      [this other] (binary-deep=Long other this))

  (binary-deep=List       [this other] (binary-deep=Long other this))
  (binary-deep=ISeq       [this other] (binary-deep=Long other this))

  (binary-deep=String     [this other] (binary-deep=Long other this))
  (binary-deep=Symbol     [this other] (binary-deep=Long other this))
  (binary-deep=Keyword    [this other] (binary-deep=Long other this))

  (binary-deep=nil        [_] false))
```

In the listings above the underlined procedures *Long=Long*, *binary-deep=Long* take part in the promotion and the actual comparison. The above code is also a pattern for all other types.

- When testing for *Long* and *Double* values equivalence, the *Double* value is checked for it's “integral” character. If it is a mathematical integer, it is downgraded to *Long* and then tested for equality with the passed *Long* value. In any other case the Double can't be an equivalent for an integral value.

```
(defn Double=Long
  [^Number this ^Number other]
  (when (DoubleMath/isMathematicalInteger
                                  (.doubleValue this))
      (= (.longValue this) (.longValue other))))
```

- The correct way to test two Doubles for equality is using the technique below.:

```
(defn Double=Double
  [^Number this ^Number other]
  (zero? (Double/compare (.doubleValue this)
                         (.doubleValue other))))
```

It is an important issue, because many programmers use the ordinary == operator to test *doubles* for equality and the == does not work properly with-*Double.NaN*.

- *BigIntegers* may be equivalent to *Longs* iff they fit in the 64-bit *Long* range. On such occasion a *BigDecimal* gets algorithmically downgraded to a Long before comparing:

```
;; BigInteger
(defn BigInteger=Long
  [^BigInteger this ^Number other]
  (when (BigInteger-in-long? this)
    (= (.longValue this) (.longValue other))))
```

- In the case of *BigIntegers* and *Doubles* the values of both types may be equivalent only if the *Double* is a mathematical integer and the *BigInteger* fits in the 64-bit *long* range. Then the values are downgraded to *Longs* and tested for equality:

```
(defn BigInteger=Double
  [^BigInteger this ^Number other]
  (when (and (DoubleMath/isMathematicalInteger
                             (.doubleValue other))
             (BigInteger-in-long? this))
    (= (.longValue this) (.longValue other))))
```

- The instances of *clojure.lang.BigInt* undergo the same rules as *java.math.BigInteger* when tested for equality against *Doubles* and *Longs*. The minor changes come out from some implementation differences between *BigInts* and *BigIntegers*. See the codes below:

```
(defn BigInt=Long
  [^BigInt this ^Number other]
  (when-not (.bipart this)
    (= (.lpart this) (.longValue other))))

(defn BigInt=Double
  [^BigInt this ^Number other]
  (when (and (DoubleMath/isMathematicalInteger
                           (.doubleValue other))
             (not (.bipart this)))
    (= (.lpart this) (.longValue other))))
```

- The equivalence of Java *BigIntegers* and Clojure *BigInts* is one of the easiest to be achieved because every *BigInt* may be converted into a *BigInteger* easily with the API call:

```
(defn BigInt=BigInteger
  [^BigInt this ^BigInteger other]
  (.equals (.toBigInteger this) other))
```

- The binary equivalence algorithm for Clojure rational numbers uses a *clojure.core/rationalize* API call to convert other numbers, namely *BigDecimals* and *Doubles* to *Ratio* instances and then to perform the equality test:

```
(binary-deep=Double [this other]
    (.equals this (rationalize other)))

(binary-deep=BigDecimal [this other]
    (.equals this (rationalize other)))

(binary-deep=Ratio [this other]
    (or (ref= this other) (.equals this other)))
```

Last, but not least there are the selected implementation details of the *kongra.bigdec* library, that is used as a tool in the equivalence realization.

Testing two instances of *java.math.BigDecimal* for equality requires using the *compareTo* method, due to similar reasons that were mentioned when describing deep-hash implementation for *BigDecimals* earlier in this section:

```
(defn =
  ([this] true)
  ([this other] (zero? (.compareTo (cast[14] this)
                                   (cast[14] other))))
  ([this other & more]
     (if (BD/= this other)
       (if (next more)
         (recur other (first more) (next more))
         (BD/= other (first more)))

       false)))
```

Testing *java.math.BigDecimals* for their integer equivalence uses the following procedure before verifying the actual value equality:

```
(defn integer?
  ([this ^Long signum]
    (let [this (cast[14] this)]
      (or (zero? signum)
          (<= (long (.scale this)) 0)
          (<= (long (.. this
                        stripTrailingZeros scale)) 0))))

  ([this]
     (let [this (cast[14] this)]
       (BD/integer? this (long (.signum this))))))
```

7 Selected Implementation Details for Reference Types

The structural and semantic equivalence implementation for reference types refers mostly the collections. For sequences the following Java procedure is used as a utility mechanism to compute a the *deep-hash* value for a sequence[15]. Please, note the *deep-hash* dependency passed as a procedure parameter:

14 The *kongra.bigdec/cast* operator is used to convert values of various types into *java.math.BigDecimal*. It's protocol-based implementation is not presented in the paper.

15 A similar mechanism runs for the *java.util.List* instances. The only difference is the use of iterator there.

```
  public static int calculate(ISeq coll, IFn deepHash) {
    if (null == coll) {
      return 0;
    }
    ISeq seq = coll.seq();
    if (null == seq) {
      return 0;
    }
    int result = 1;
    do {
      Object element = seq.first();
      result = 31 * result + calculate(element,
                                       deepHash);
      seq = seq.next();
    }
    while (null != seq);
    return result;
  }
```

The above algorithm closely mimics similar algorithms belonging both to the Java standard library[16] as well as the Clojure language library and run-time[17].

Thus the implementation of deep-hash for objects sequential in nature:

```
(extend-protocol WithDeepHash
  java.util.List
  (deep-hash [this]
    (DeepHash/calculate this ^IFn deep-hash))

  clojure.lang.ISeq
  (deep-hash [this]
    (DeepHash/calculate this ^IFn deep-hash))

  kongra.core.Pair
  (deep-hash [this]
    (DeepHash/calculate this ^IFn deep-hash)))
```

Strings are also treated as sequences (of Characters, thus integrals):

```
(extend-protocol WithDeepHash
  java.lang.String
  ;; String is treated as a sequence of chars
  (deep-hash [this]
    (DeepHash/calculate (seq this) ^IFn deep-hash)))
```

16 E. g. the class *java.util.Arrays* and the hash implementation for arrays of primitives.

17 E. g. the class *clojure.lang.ASeq* and it's implementation of *hashCode* method.

Testing for the deep binary equivalence of sequential containers is being realized in the form of a procedure below. As it was in the case of deep hashes realizations, it's shape was also influenced by related mechanisms in the Clojure standard library. The mechanism is used also for Strings – as it was stated earlier they are treated as sequences.

```
(defn ISeq=ISeq
  [this other]
  (loop [this  (seq this)
         other (seq other)]

    (cond (nil? this) (nil? other)
          (nil? other) false
          (not (binary-deep= (first this)
                             (first other))) false
          :else (recur (next this) (next other)))))
```

Another class of container types are the sets and maps, in other words the associative collections. The implementation of deep equivalence is present in the described library, but it's full presentation goes deeply beyond the scope of this paper.

One final mark we must stop at is the tag-based typing. All custom Java classes as well as Clojure records and types fall within the category of tagged types. As it was mentioned in the initial sections of the article, the deep, structural equivalence implemented by the author and described here is targeted towards duck-typed systems, so the library does not cover the tag-based typing at all.

References

1. Chugh R., Rondon P.M., Jhala R., 2012, *Nested refinements: a logic for duck typing*, POPL '12 Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pp. 231-244
2. Gosling J., Joy B., Steele G., Bracha G., 2005, *The Java^{TM} Language Specification Third Edition*, ISBN 0-321-24678-0, available at the Oracle Technology Network (2014) http://docs.oracle.com/javase/specs/
3. Langer A., 2002, *Secrets of equals() - Part 1*, www.angelikalanger.com/Articles/JavaSolutions/SecretsOfEquals/Equals.html
4. Venners B., 2002, *Josh Bloch on Design, instanceof versus getClass in equals Methods*, JavaWorld January 4, 2002, www.artima.com/intv/bloch17.html
5. Halloway S., 2009: *Programming Clojure*, ISBN: 978-1-93435-633-3, The Pragmatic Bookshelf
6. Emerick Ch., Carper B., Grand Ch., 2012, *Clojure Programming*, O'Reilly Media Inc., ISBN: 978-1-449-39470-7

7. Kiczales G., Lamping J., Mendhekar A., Maeda Ch., Lopes C., Loingtier J-M., Irwin J., 1997, *Aspect-Oriented Programming*, Proceedings of the European Conference on Object-Oriented Programming, vol.1241. pp. 220–242
8. Bloch J., 2008, *Effective Java^{TM} Second Edition*, Addison-Wesley, ISBN-13: 978-0-321-35668-0
9. Steele G.L., 1990, *Common Lisp the Language, 2nd Edition*, Digital Press
10. Liskov B., Wing J. 1999, *Behavioral Subtyping Using Invariants and Constraints*, CMU technical report, available as: http://reports-archive.adm.cs.cmu.edu/anon/1999/CMU-CS-99-156.ps
11. IEEE Computer Society, 2008, *IEEE Standard for Floating-Point Arithmetic*. IEEE. doi:10.1109/IEEESTD.2008.4610935. ISBN 978-0-7381-5753-5. IEEE Std 754-2008
12. *Java class java.math.BigDecimal source code*, available (2014) as http://developer.classpath.org/doc/java/math/BigDecimal-source.html
13. Bloch J., Gafter N., 2005, *Java¿ Puzzlers: Traps, Pitfalls, and Corner Cases*, Addison-Wesley Professional, ISBN-10: 032133678X, ISBN-13: 978-0321336781
14. Martin R.C., 2002, *The Principles, Patterns, and Practices of Agile Software Development*, Robert C. Martin, Prentice Hall, available as (2014) http://objectmentor.com/resources/articles/visitor.pdf
15. Google, 2014, *Guava Libraries*, https://code.google.com/p/guava-libraries/