REPO: HIGH-LEVEL PERSISTENCE LAYER FOR CLOJURE

Konrad Grzanek

IT Institute, Academy of Management, Łódź, Poland
kgrzanek@swspiz.pl, kongra@gmail.com

Abstract

One of our previous works was dedicated to creating an effective no-SQL database solution for Clojure. The solution called Store still missed a high-level data definition language, the concept of objects, complex types and other programmers' productivity increasing features of a production-quality database product. The paper presents a new DSL embedded in Clojure that addresses all those expectations and works seamlessly with the site functional language.

Keywords: Functional programming, Lisp, Clojure, database

1 Introduction

The advent of functional programming style and growing popularity of JVM Lisp dialect called Clojure [4, 5] are certain catalysts for the research and engineering activities in the field of production-ready database products for the platform. One of the author's previous works was dedicated to creating an effective no-SQL database solution for this language. The solution proposed by us and described in [2] called ***Store*** is:

- a log-file based storage using Berkeley DB Java Edition [6] library
- running in-process with the client application
- possessing a simple Clojure API

Despite it's simplicity and ease of use the ***Store*** still misses few important elements making the database a production-quality solution:

- It does not posses a clear notion of persistent objects, nor an established representation of object references. All it has is a key to value mapping abstraction. In general a ***store*** is just an ***index*** with associated ***Store*** types.
- Besides some basic ***Store*** types it lacks a nominal type system [1] that would make the integration with the host language more natural and that would increase programmer's productivity.

- Following the lack of the nominal type system there is an absence of Data Definition Language (DDL) for ***Store***. This deficiency is especially painful for the programmers accustomed to traditional relational or object-oriented (network) database models.
- The mentioned weaknesses effectively prevent using polymorphic procedures based on nominal types and clear object definition, namely ***multimethods*** [4] in Clojure.

Due to the reasons pointed out above from the beginning of work on *Store* it was pretty apparent that the library and storage layer would be only the most low-level database solution and that the whole product would gain the expected production-level quality only after adding layers of abstraction above.

The solution is called a ***Repo***, following the convention of naming the database layers used previously by the author ([3]). This paper presents it's most important features and some internals of the implementation. The article is organized as follows. In the beginning the ***Repo***'s type system will be described. Then we will take a look at persistent objects, their representation and storage. After that the persistent object properties will be discussed together with their accessors. Finally some derived features will be presented like the sequential objects and properties scan and various utilities.

2 Type system

The ***Repo***'s name based (***nominal***) type system embraces the basic semantics of traditional object-oriented programming languages. Being and indirection layer embedded in Clojure, it is dynamically typed like it's host language.

There are two categories of named types in ***Repo***. The first category are the ***Store types***. These are the most fundamental types that may be treated as ***basic*** or ***primitive***, because they appear in the low level data access layer. Some of them were described in [2], namely:

:str (java.lang.String)	String literal type
:boolean (:bool, java.lang.Boolean)	truth/falsity type
:byte (java.lang.Byte) ***:short (java.lang.Short)*** ***:char (java.lang.Character)*** ***:int (java.lang.Integer)*** ***:long (java.lang.Long)***	Integral types
:float (java.lang.Float) ***:double(java.lang.Double)***	Floating-point types
:bigint (java.math.BigInteger)	Large integral type
:bitset (java.util.BitSet)	Bit set type
:timestamp (:time, java.sql.Timestamp)	Time value type
:int-pair ***:long-pair***	Type of pairs (instances of ***kongra.core.Pair*** custom utility class) of primitive integral values

As can be seen the ***Store types*** are identified by their names, and they are essentially Clojure ***keywords***. For the types mentioned above there is also an alternative identification scheme using Java platform types. According to this an example store mapping strings to integers may be accessed like:

```
(store <name> :str :int)
```
or
```
(store <name> String Integer)
```

It is worth mentioning the ***Store*** offers a service (actually an API) that allows the users to define their own primitive types. The usage is pretty straightforward and it boils down to registering a ***serializer*** and ***desertializer*** procedures for a specified set of Clojure keywords or Java classes denoting the primitive type:

```
(require '[kongra.store :as DB])

(DB/register-converters

 (fn [^RObj obj]                 ;; serializer
   (Bits/longToBytes
     (.identity (RTools/assertStorable obj))))

 (fn ^RObj [bytes]               ;; deserializer
   (R-obj (Bits/getLong bytes 0)))

   :R-obj)                       ;; new Store type name
```

The listing at the bottom of the previous page shows the way a primitive ***Store*** type of name ***:R-obj*** is being defined by using the mentioned extension mechanism.

Repo introduces the following basic types:

:R-obj	A type of the persistent ***R-object*** reference. The persistent objects are the major kinds of data managed by ***Repo***. See the listing at the bottom of the previous page and Section 3.
:R-type	A persistent meta-type for ***Repo*** types that are the first-class objects in our language.
:R-any	A type of any object that may undergo serialization/deserialization to ***java.lang.String***.
:R-seq, :R-list	A type of sequences of persistent objects (***R-objects***)
:R-set	A type of sets of persistent objects (***R-objects***)
:R-pair	A type of pairs (instances of ***kongra.core.Pair*** custom utility class) of persistent objects (R-objects)
:R-map	A type of maps ***:R-obj → :R-obj*** (where persistent objects are both keys and values)
:R-bindings	A type of maps ***:str → :R-obj*** (with ***java.lang.String*** keys and persistent objects values)

Besides the primitive ***Store types*** there are the complex types that are the core of the whole ***Repo*** type system. These complex types are called ***R-types***.

The ***R-types*** are first-class objects in Clojure. This also refers to the basic ***Store types***, as ***keywords*** and class names that denote them are the first-class objects is the host language. An ***R-type*** is essentially just a named (symbolic) wrapper around an unsigned 16-bit integer (thus in inclusive range 0 .. 65535). So the limiting number of types in any system implemented using the ***Repo*** is 65536. We consider the number to be sufficient in production environments and – as will be stated further (in Section 3) – it opens the way for the effective persistent objects encoding.

R-types may participate in ***multiple inheritance*** and there is no $\top$ (***top***) ***type***, albeit defining a custom ***subclassing*** hierarchy root is perfectly possible only not mandated by the language itself.

The ***R-types*** definition language is given by the following rules:

<R-type definition> ::= (**R-deftype** *<name>* [*<base-types>*] *<properties>*)
<base-types> ::= ε | *<base-type>* *<base-types>*
<properties> ::= ε | *<property>* *<properties>*
<property> ::= (*<property name>* *<property type>*)

<*name*> ::= instance of ***clojure.core.Symbol***
<*property name*> ::= instance of ***clojure.core.Keyword | clojure.core.Symbol***
<*property type*> ::= a valid primitive ***Store type***
<*base-type*> ::= a symbol denoting already defined ***R-type***

To illustrate the way ***R-types*** are defined, let's take a look at the following example:

```
(use 'kongra.repo)

(R-deftype WithSrcInfo []
  (:src-file :R-obj)
  (:src-line Integer)
  (:src-pos  Integer))

(R-deftype Named [WithSrcInfo]
  (:name String))

(R-deftype Obj [])

(R-deftype CompilationUnit [Named Obj]
  (:types :R-seq)
  (:import-clauses :R-seq))
```

Above there are the definitions of ***R-types*** being a part of the Java persistent model in our software modeling and analysis tool. The types ***WithSrcInfo*** and ***Obj*** do not have any super-type, while the ***CompilationUnit*** has two base types (with respect to the ***subclassing*** relation, not ***subtyping***), and so it can be read that the ***CompilationUnit*** is a named object with the properties ***:types*** and ***:import-clauses***.

As it was stated earlier, the type system is dynamic and strong (no implicit casts are possible), following the host language in this manner. But the property types specification concerns solely the primitive ***Store types*** and not the ***R-types***. One can set an arbitrary ***R-type*** instance for the ***:src-file*** in a ***WithSrcInfo*** (or derivative) persistent object, and the only verification will be asserting the ***:R-obj*** primitive ***Store type*** of the value being set.

The Repo API offers the following predicates related to the ***subclassing*** semantics:

- (***R-isa?*** <*parent-type*> <*type*>) → ***true*** if and only if the ***type*** is a subclass (direct or indirect) of the ***parent-type***.
- (***R-instance?*** <*type*> <*obj*>) → ***true*** if and only if the persistent ***obj*** is of the *R-type* ***type.***

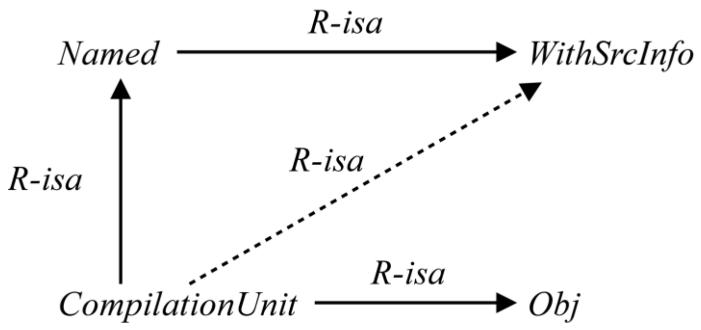

Figure 1. Example subclassing diagram

The inheritance diagram corresponding to the last example is shown above at Figure.1. The ***R-isa*** relation is closely coupled with the ***R-isa?*** predicate; the ***R-isa*** holds whenever ***R-isa?*** evaluates to true.

3 R-objects

The persistent ***Repo*** objects are called the ***R-objects***. They are related to ***R-types*** exactly the same way the Java object is related to it's class.

An ***R-object*** is an entity consisting of an identifier (an object of ***kongra.repo.RID*** class) bundled together with it's ***R-type***. The identifier is a wrapper around an unsigned 48-bit integer, so the correct ***RIDs*** values range inclusively from 0 up to 281474976710655. Because the ***R-type*** is represented by a 16-bit unsigned integer, the full ***R-object*** integral representation takes exactly 64 bits, that is a storage of a single ***java.lang.Long*** value. Figure 2 shows the way the persistent ***R-object*** looks like internally.

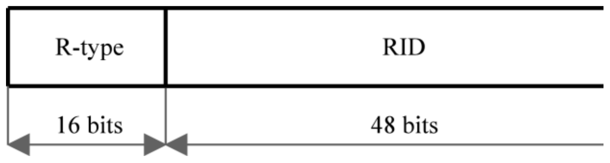

Figure 2. R-object representation layout

The most obvious way to get a reference to an ***R-object*** is to call it's ***R-type*** with the ***RID*** argument:

```
(CompilationUnit 5)
```

This form returns a ***CompilationUnit*** instance of ***RID*** 5. To create a new instance the ***R-type*** should be called either with no arguments like

```
(CompilationUnit)
```

or with the properties and corresponding values:

```
(CompilationUnit :name "src/Program.java")
```

In both cases a new ***RID*** would be generated by an underlying ***Store sequence***.

4 Storage

The way ***Repo*** approaches the persistent ***R-objects*** storage and life-cycle management is unlike the traditional ways that can be met in other database solutions. In the first place it must be underlined that the expressions like

```
(<R-type> <RID>)
```

return references to ***R-objects*** that may be either existent or not. More broadly speaking, the ***Repo*** does not manage the ***RIDs'*** life-cycle by storing them on ***R-object***'s creation and deleting on removal. The only place where the persistent changes related to ***RIDs*** are made during an ***R-object*** creation is the underlying ***Store sequence*** mechanism. Because a sequence is unidirectional, nothing related to the identifiers happens when a persistent ***R-object*** is being removed. This approach is somewhat counterintuitive for the programmers accustomed to the relational model in particular, but it has got it's advantages:

- ***R-objects*** creation and deletion is very cheap both in terms of storage size and performance,
- The approach exhibits the dynamic characteristics staying on track with the philosophy of the host language

The storage mechanisms start playing vital role when storing the property values of ***R-objects***. When someone wants to set a property value on a persistent object he has to call ***R-assoc*** procedure properly. The procedure usage is as follows:

```
(R-assoc <obj>
         <property 1> <value 1>
         ...
         <property N> <value N>)
```
[1][2]

Every property of a specified ***R-type*** has an associated ***store*** that maps a ***RID*** onto the property value. Accessing the property of a persistent object always requires only an access to this store and so it is fine-grained, fetching the complete object is not necessary. It also creates an opportunity to locate the physical data of properties on different partitions, hosts etc.

The most simple way to get a value of a property is to use an R-object as a map:

```
(def cu (CompilationUnit 5)) ;; cu denotes a CompilationUnit
                             ;; of RID 5.

(cu :name)                   ;; evaluates to the :name of
                             ;; an object represented by cu
```

When an entry in the underlying ***store*** for a property of a particular type is missing, the property value is ***nil*** (***null*** in Java). When one sets a property value to ***nil***, the appropriate entry is being removed (if present) from the ***store***. This fact in the correlation with the fine-grained way the ***Repo*** manages property values makes the whole solution especially useful for managing sparse data.

Following that, the ***R-object's*** deletion is solely a process of setting all property values to nil for the ***R-object's RID*** – effectively removing all entries for ***RID*** key from all stores of all ***R-object's*** properties. Yet – as it was mentioned earlier when discussing the Repo storage principles – the removed R-object is still operational:

```
(R-del cu)           ;; deletes the object represented by cu

(R-assoc cu :name "some/other.name")
                     ;; cu is perfectly valid for further
                     ;; operations
```

1 The procedure closely resembles ***clojure.core/assoc***. It is worth noting that the persistent ***R-objects*** support the maps semantics with respect to the querying operations by implementing ***clojure.lang.ILookup***.

2 Setting the property values is also possible on ***R-object***'s creation, when calling the proper constructor variant, e.g. `(CompilationUnit :name "src/Program.java")` associates the property ***:name*** with the specified String while creating a new persistent ***CompilationUnit*** representation.

5 Accessors

The notion of property accessors is not uncommon in the Lisp world. One of the most significant works using this term and semantics was the ***Common Lisp Object System*** (***CLOS***) described in [7]. In general an ***accessor*** is a procedural mechanism for accessing values of object's properties. Java programming language comes with a similar idea expressed by ***POJO*** (Plain Old Java Objects) getters and setters wrapping the access to fields ([8]). ***Repo*** keeps this semantics introducing it's own implementation that is to be described in this section.

There are two kinds of accessors in ***Repo***:

1. ***Readers*** – procedures created by the following λ-expression:
   (fn *[obj type property] ...)*
2. ***Writers*** – procedures of shape: ***(fn*** *[obj type property value] ...)*

where ***obj*** is the R-object on which the access is to be made, ***type*** is the exact ***R-type*** of the ***R-object***, ***property*** is the name (symbol, keyword) of the property to be accessed and value of the ***writer*** is the property value to be written to the store for ***<type, property>*** pair.

The ***Repo*** API allows to define a ***reader*** by calling ***R-defreader*** like below:

```
(R-defreader CompilationUnit :name

 (fn [obj type property]
   ...))
```

The above expression defines a reader procedure for the ***CompilationUnit R-type*** and ***:name*** property. Similarly the form

```
(R-defwriter CompilationUnit :name

 (fn [obj type property value]
   ...))
```

is a definition of a ***writer*** for the specified ***R-type*** and property. Both accessors may be defined in one place, by making a simplified ***R-defaccessor*** call:

```
(R-defaccessor CompilationUnit :name

 (fn [obj type property] ...)

 (fn [obj type property value] ...))
```

There are the following utilities that may simplify implementing the accessors' bodies:

R-this	A predefined accessor (an object playing the role of both reader and writer) that represents the current accessing mechanism registered for the ***R-type*** of the ***R-object*** and the specified property.
R-super	An accessor representing the current accessing mechanism of the super-type. When using the multiple inheritance property of ***Repo*** type system must be taken into account.
R-raw-reader ***R-raw-writer*** ***R-raw***	Reader, writer and accessor representing the default ***Store***-based access mechanism.
R-get	A procedural wrapper around property reading. If there is a necessity of accessing the ***R-object***'s property value using an explicitly specified reader, this procedure can be used, e. g.: `(R-get cu R-raw :name "Default name value")` The example above returns a value of the ***:name*** of the ***R-object*** represented by the ***cu*** symbol. Additionally if the read ***:name*** is ***nil***, the passed default string will be used as the result. Mimics the ***clojure.core/get*** procedure.
R-assoc	Writing procedure described above also supports accessors. See the example below: `(R-assoc cu R-super :name "some/file.c")` where the super-type accessor (writer) is used to write the ***:name*** of the ***R-object*** represented by the ***cu*** symbol.

6 Running across R-objects

Due to to the principles of ***Repo*** storage described in Section 4, there is no possibility to perform a sequential run across the ***R-objects*** of a given ***R-type***. Instead, there is a way to access ***R-objects*** together with the specified property. Sequential runs in ***Repo*** apply to the ***stores*** that underly the properties. Speaking more precisely, they are made across the ***store entries*** contained in the particular store, rather than ***RIDs*** or ***R-objects*** themselves. The following procedures are the crucial ones here:

R-entries	Returns a collection of ***pairs*** (instances of ***kongra.core.Pair*** utility class) representing all entries for the passed ***R-type*** and property. An additional parameter ***with-subtypes?*** (***true*** by default) means the resolution will be performed also for subtypes of ***R-type***. Setting it to ***false*** or ***nil*** narrows the process only to the passed ***R-type*** argument. Requires a cleanup context[3] to exist on the run-time.
R-do-entries	Executes the unary function passed as the argument on every pair belonging to the collection that represents all entries for the passed ***R-type*** and property. Executes within it's own own cleanup context.
R-map-entries	Returns a collection of entries in ***R-type*** and property mapped onto f. f is expected to accept a single pair argument. An additional parameter allows to specify whether or not the collection is to be lazy. Non-laziness is the default. If the laziness is requested, a cleanup context is required.

7 Summary

Presented database solution is a thin yet highly valuable language layer built on top of a very effective in-process, no-SQL data storage and implemented as a domain specific language embedded in a host language, Clojure in this case. The design phase was in fact dedicated to accomplish all the following goals:

3 The cleanup context mentioned in [2]. It is a syntactic and semantic construct implemented as a Lisp macro ([9]) and using dynamic variables that allows to effectively and safely manage persistent and – in general – vulnerable resources that require an explicit fetch/release, like the memory, networking connections, files system handles. The basic usage is simple: (***doclean*** …) where the body (represented by …) will be executed within the cleanup context.

- Ease of use from the programmer's point of view implying minimalism of the API
- Power of the means of expression in the sub-language assuming the orthogonality an completeness of all operators/procedures
- Performance4 and a relative “low weight” of the layer

These initial assumptions influenced not only the ***Repo*** design but also implementation phase. To what extent the author managed to accomplish the initially decreed purpose is an open question targeted towards all the potential future users.

Currently ***Repo*** is used as a vital part of software analysis and reasoning tool. Any future enhancements are planned to be done on demand.

References

1. Pierce B.C., 2002: *Types and programming languages,* pp. 1–632. MIT Press
2. Grzanek K., 2010, *Store: Embedded Persistent Storage For Clojure Programming Language,* Journal of Applied Computer Science Methods No. 1 Vol. 2, pp. 83
3. Grzanek K., 2009: *Realization of The Design Patterns Occurrences Recognition System with Static Analysis Methods, PhD Thesis*, Department of Computer Engineering, Czestochowa University of Technology, pp.1–192
4. Halloway S., 2009: *Programming Clojure*, ISBN: 978-1-93435-633-3, The Pragmatic Bookshelf
5. Clojure Website 2012: http://clojure.org
6. Oracle Corp. 2012, *Berkeley DB Java Edition Architecture*, http://www.oracle.com/ technetwork/products/berkeleydb/learnmore/bdb-je-architecture-whitepaper-366830.pdf
7. Kiczales G., Rivieres J., Bobrow D.G., 1991, *The Art of the Metaobject Protocol,* MIT Press, ISBN 0-262-61074-4
8. Oracle Corp. 2012*, Java Language and Virtual Machine Specifications,* http://docs.oracle.com/javase/specs/
9. Graham P., 1993, *On Lisp - Advanced Techniques for Common Lisp.* Prentice Hall

4 The performance considerations and run-time benchmarking results go far beyond the scope of this paper despite their importance.