Reversible Arithmetic on Collections

The Team of Fu

September 22, 2013

Contents

Remark This is a literate program. [1] Source code *and* PDF documentation spring from the same, plain-text source files.

1 Introduction

We often encounter data records or rows as hash-maps, lists, vectors (also called *arrays*). In our financial calculations, we often want to add up a collection of such things, where adding two rows means adding the corresponding elements and creating a new virtual row from the result. We also want to *un-add* so we can undo a mistake, roll back a provisional result, perform a backfill or allocation: in short, get back the original inputs. This paper presents a library supporting reversible on a large class of collections in Clojure.[2]

[1] `http://en.wikipedia.org/wiki/Literate_programming`.
[2] `http://clojure.org`

2 Mathematical Background

Think of computer lists and vectors as *mathematical vectors* familiar from linear algebra:[3] ordered sequences of numerical *components* or *elements*. Think of hash-maps, which are equivalent to *objects* in object-oriented programming,[4] as sparse vectors[5] of *named* elements.

Mathematically, arithmetic on vectors is straightforward: to add vectors, just add the corresponding elements, first-with-first, second-with-second, and so on. Here's an example in two dimensions:

$$[1, 2] + [3, 4] = [4, 6]$$

Clojure's *map* function does mathematical vector addition straight out of the box on Clojure vectors and lists. (We don't need to write the commas, but we can if we want – they're just whitespace):

```
(map + [1 2] [3 4])
```

```
==> [4 6]
```

With Clojure hash-maps, we can add corresponding elements via *merge-with*

```
(merge-with + {:x 1, :y 2} {:x 3, :y 4})
```

```
==> {:x 4, :y 6}
```

The same idea works in any number of dimensions and with any kind of elements that can be added (any *mathematical field*:[6] integers, complex numbers, quaternions, many more.

Now, suppose you want to *un-add* the result, `[4 6]`? There is no unique answer. All the following are mathematically correct:

$$\begin{aligned}
[-1, 2] + [5, 4] &= [4, 6] \\
[0, 2] + [4, 4] &= [4, 6] \\
[1, 2] + [3, 4] &= [4, 6] \\
[2, 2] + [2, 4] &= [4, 6] \\
[3, 2] + [1, 4] &= [4, 6]
\end{aligned}$$

and a large infinity of more answers.

[3] `http://en.wikipedia.org/wiki/Linear_algebra`
[4] `http://en.wikipedia.org/wiki/Object-oriented_programming`
[5] `http://en.wikipedia.org/wiki/Sparse_vector`
[6] `http://en.wikipedia.org/wiki/Field_(mathematics)`

3 A Protocol for Reversible Arithmetic

Let's define a protocol for *reversible arithmetic in vector spaces* that captures the desired functionality. We want a *protocol* – Clojure's word for *interface*,[7] because we want several implementations with the same reversible arithmetic: one implementation for vectors and lists, another implementation for hash-maps. Protocols let us ignore inessential differences: the protocol for reversible arithmetic on data-rows-as-mathematical-vectors should be the same for all collection types.[8]

Name our objects of interest *algebraic vectors* to distinguish them from Clojure's existing *vector* type. Borrowing an idiom from C# and .NET, name our protocol with an initial *I* and with camelback casing.[9] Don't misread *IReversibleAlgebraicVector* as "irreversible algebraic vector;" rather read it as "Interface to Reversible Algebraic Vector," where the "I" abbreviates "Interface."

We want to add, subtract, and scale our reversible vectors, just as we can do with mathematical vectors. We include inner product, since it is likely to be useful. Though we don't have an immediate scenario for subtraction and inner product, the mathematics tells us they're fundamental. Putting them in our design *now* affords two benefits:

1. when the need arises, we won't have to change the code
2. their existence in the library may inspire usage scenarios

Remark The choice to include operations in a library in the absense of scenarios is a philosophical choice,[10] perhaps more akin to *Action-Centric design* or *proactive* design as opposed to *Rational* or *minimalist* design. The former philosophy promotes early inclusion of facilities likely to be useful, where as the latter demands ruthless reduction of facilities not known to be needed. The former is based on intuition, judgment, and experience, and the latter is based on ignorance of the future. We thus prefer the former.

Finally, we need undo and redo, the differentiating features of reversible algebraic vectors. Here is our protocol design:

[7] `http://en.wikipedia.org/wiki/Interface_(computing)`
[8] including streams over time! Don't forget Rx and SRS.
[9] `http://en.wikipedia.org/wiki/CamelCase`
[10] `http://en.wikipedia.org/wiki/Design_philosophy`

```
(defprotocol IReversibleAlgebraicVector
  ;; binary operators
  (add   [a b])
  (sub   [a b])
  (inner [a b])
  ;; unary operators
  (scale [a scalar])
  ;; reverse any operation
  (undo [a])
  (redo [a])
)
```

3.1 Implementing the Protocol for Vectors and Lists

As a first cut, let us wrap algebraic vectors in hash-maps that contain enough information to reverse any computation.

First, define the base case: collections that hold data that can be treated as ordinary, non-reversible, algebraic vectors. What kinds of things can hold ordinary vector data? They must be things we can operate on with *map* or *merge-with* to perform the basic, vector-space operations. Therefore, they be Clojure vectors, lists, or hash-maps.

The higher-level case is to store reversing information in hash-maps along with base-data. The base data will belong to the *:data* key, by convention.

Definition 3.1 (Reversible Algebraic Vector) *A **reversible algebraic vector** is either a **base-data** collection or a hash-map containing a `:data` attribute. A base-data collection is either a Clojure vector, list, or hash-map that does not contain a `:data` attribute. If a reversible algebraic vector does contain a `:data` attribute, the value of that attribute is a base-data collection.*

Here is a *fluent* type-checking function for base-data. It either returns its input – like the *identity* function – or throws an exception if something is wrong.

```
(defn- check-data-type [that]
  (let [t (type that)]
    (if (or (= t (type []))
            (= t (type '()))  ; empty list is special
            (= t (type '(0))) ; this list is ordinary
            (and (= t (type {})) (not (contains? that :data))))
      that ; ok -- otherwose:
      (throw (IllegalArgumentException.
        (str "This type of object can't hold vector data: " t))))))
```

Now we need a way to get the data out of any reversible algebraic vector.

If the input is a hash-map, we must explicitly check for existence of *:data* so that we can tell the difference between a hash-map that has *:data* whose value is *nil*, which is an illegal case, and a hash-map that has no *:data*, a legal case. We cannot simply apply the keyword *:data* to the candidate reversible vector because that application would produce *nil* in both cases. Instead, we apply *:data* to the candidate after checking for existence of the key, and then we apply *check-data-type*, defined above.

```
(defmulti  get-data type)
(defmethod get-data (type [])   [that] that)
(defmethod get-data (type '())  [that] that)
(defmethod get-data (type '(0)) [that] that)
(defmethod get-data (type {})   [that]
  (if (contains? that :data)
    (check-data-type (:data that))
    that))
(defmethod get-data :default    [that]
  (throw (IllegalArgumentException.
    (str "get-data doesn't like this food: " that))))
```

Now we write a test for all these cases. We require *IllegalArgumentExceptions* for bases-data blocks that are not vectors, lists, or hash-maps or base-data blocks that contain reversible-vectors: our design does not want to nest such vectors.

```
(deftest get-data-helper-test
  (testing "get-data-helper"
    ;; Negative tests
    (are [val] (thrown? IllegalArgumentException val)
      (get-data 42)
      (get-data 'a)
      (get-data :a)
      (get-data "a")
      (get-data \a)
      (get-data #inst "2012Z")
      (get-data #{})
      (get-data nil)
      (get-data {:data 42 })
      (get-data {:data 'a })
      (get-data {:data :a })
      (get-data {:data "a"})
      (get-data {:data \a })
      (get-data {:data #inst "2012Z"})
      (get-data {:data #{} })
      (get-data {:data nil })
      (get-data {:data {:data 'foo} })
    )
    ;; Positive tests
    (are [x y] (= x y)
          [] (get-data  [])
         '() (get-data '())
          {} (get-data  {})

         [0] (get-data  [0])
        '(0) (get-data '(0))
      {:a 0} (get-data  {:a 0})

       [1 0] (get-data  [1 0])
      '(1 0) (get-data '(1 0))
 {:a 0 :b 1} (get-data  {:b 1 :a 0})

        [42] (get-data {:a 1 :data [42]})
       '(42) (get-data {:a 1 :data '(42)})
     {:a 42} (get-data {:a 1 :data {:a 42}})

          [] (get-data {:a 1 :data []})
         '() (get-data {:a 1 :data '()})
          {} (get-data {:a 1 :data {}})
    )
))
```

To implement the protocol, we will need multimethods that dispatch on the types of the base data. There is an example of this above in get data; let's follow it to build add-data:

```
(defn two-types [a b]
(defmulti  add-data two-types)
(defmethod add-data (type [])   [that] that)
(defmethod add-data (type '())  [that] that)
(defmethod add-data (type '(0)) [that] that)
(defmethod add-data (type {})   [that]
  (if (contains? that :data)
    (check-data-type (:data that))
    that))
(defmethod add-data :default    [that]
  (throw (IllegalArgumentException.
    (str "get-data doesn't like this food: " that))))



(defrecord ReversibleVector [a-vector]
  IReversibleAlgebraicVector
  (add   [a b] {:left-prior a, :right-prior b,
                :operation 'add, :data (map + (get-data a)
                                              (get-data b))})
  (sub   [a b] nil)
  (inner [a b] nil)
  (scale [a scalar] nil)
  (undo  [a] nil)
  (redo  [b] nil))
```

4 Unit-Tests

```
(ns ex1.core-test
  (:require [clojure.test :refer :all]
            [ex1.core     :refer :all]))
```

5 REPLing

To run the REPL for interactive programming and testing in org-mode, take the following steps:

1. Set up emacs and nRepl (TODO: explain; automate)
2. Edit your init.el file as follows (TODO: details)
3. Start nRepl while visiting the actual |project-clj| file.
4. Run code in the org-mode buffer with `C-c C-c`; results of evaluation are placed right in the buffer for inspection; they are not copied out to the PDF file.