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

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

paper presents a library supporting reversible on a large class of collections in Clojure.[2]

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 them, 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 in Clojure):

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

==> [4 6]
```

With Clojure hash-maps, 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.

[2] http://clojure.org
[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)

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 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 is the same for all compatible 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 "I Reversible Algebraic Vector", i.e., "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. Include inner product, since it is likely to be useful. Though we don't have immediate scenarios for subtraction, scaling, 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

[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`

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 *Hyper-Rationalist* or *minimalist* design. The former philosophy promotes early inclusion of facilities likely to be useful or inspirational, whereas the latter philosophy demands ruthless rejection of facilities not known to be needed. Both buy into to removing facilities *known to be not needed*, of course. The former philosophy is based on intuition, judgment, and experience, and the latter philosophy embraces ignorance of the future as a design principle. We thus prefer the former.

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

```
(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])
)
```

4 Implementing the Protocol

4.1 Defining r-vectors and a-vectors

What things represent algebraic vectors? Things we can operate on with *map* or *merge-with* to perform basic vector-space operations. Therefore, they must be Clojure vectors, lists, or hash-maps.

The higher-level case wraps reversing information in a hash-map along with base-case algebraic vector data. The base data will belong to the *:a-vector* key, by convention.

[10] `http://en.wikipedia.org/wiki/Design_philosophy`

Definition 4.1 (Reversible Algebraic Vector (r-vector)) *A* ***reversible algebraic vector*** *or* ***r-vector*** *is either an algebraic vector, i.e.,* ***a-vector****, or a hash-map containing an* `:a-vector` *attribute. An a-vector is either a Clojure vector, list, or hash-map that does not contain a* `:a-vector` *attribute. If an r-vector does contain a* `:a-vector` *attribute, the value of that attribute must be an a-vector.*

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

```
(defn- check-a-vector [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 :a-vector))))
      that ; ok -- otherwise:
      (throw (IllegalArgumentException.
        (str "This type of object can't hold vector data: " t))))))
```

4.2 Fetching a-vector Data

Now we need a way to get a-vector data out of any r-vector. If the r-vector is an a-vector, just return it. Otherwise, if the r-vector is a hash-map, fetch and check the value of the :a-vector attribute. In all other cases, reject the input.

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

```
(defmulti  get-a-vector type)
(defmethod get-a-vector (type [])   [that] that)
(defmethod get-a-vector (type '())  [that] that)
(defmethod get-a-vector (type '(0)) [that] that)
(defmethod get-a-vector (type {})   [that]
  (if (contains? that :a-vector)
    ;; throw if the contained a-vector is illegal
    (check-a-vector (:a-vector that))
    ;; otherwise, just return the input
    that))
(defmethod get-a-vector :default    [that]
  (throw (IllegalArgumentException.
    (str "get-a-vector doesn't like this food: " that))))
```

4.3 Testing the Definition

Now we write a test for all these cases. We require *IllegalArgumentExceptions* for inputs that are not vectors, lists, or hash-maps and for a-vector hash-map values that contain r-vectors: our design does not nest r-vectors.

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

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

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

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

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

4.4 Implementing the Protocol

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 :a-vector)
    (check-a-vector (:a-vector that))
    that))
(defmethod add-data :default    [that]
  (throw (IllegalArgumentException.
    (str "get-a-vector doesn't like this food: " that))))



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

5 Unit-Tests

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

6 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.