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,

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

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]

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. *Add* should be multiary, because that's intuitive. Multiary functions are written with ampersands before all the optional parameters, which Clojure will package in a sequence. *Sub* should be binary, because multiary sub is ambiguous. Include inner product, because 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

[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`

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 *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 relies on intuition, taste, 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 & more])
  (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.*

4.2 Checking the Definition

Here is a type-checking function for a-vector data. This function is private to the namespace (that's what the '-' in *defn-* means). It takes a single parameter named *that*. It promotes *fluent* or function-chaining style by being, semantically, the identity function – it either returns its input or throws an exception if something is wrong.

```
(defn- check-a-vector [that]
  (if (or (list? that)
          (vector? that)
          (and (map? that) (not (contains? that :a-vector))))
      that ; ok -- otherwise:
      (throw (IllegalArgumentException.
        (str "This type of object can't hold vector data: "
             (type that))))))
```

4.3 Fetching *a-vector* Data

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 with an exception.

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)
;; empty list has its own type, but it is still a list
(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))))
```

4.4 Unit-Testing *get-a-vector*

We require *IllegalArgumentExceptions* for inputs that are not a-vectors and for r-vectors that contain r-vectors: our design does not nest r-vectors.

Let's make a couple of test sets for data that should be accepted and rejected immediately. Creating new tests is as easy as adding new instances to these sets.[11] We include some types that may not be acceptable for arithmetic. Here, we are just testing structure.

[11] this is a white lie, as we see below; but the technique of data-driven testing is worth illustrating.

```
(def ^:private good-ish-test-set
  '([] () {} [0] (0) {:a 0} [1 0] (1 0) {:a 0, :b 1}
    [true] [false] [nil] (true) (false) (nil)
    {:a true} {:a false} {:a nil} )  )

(def ^:private bad-ish-test-set
  '(42 'a :a "a" \a #inst "2012Z" #{} nil true false
   {:a-vector 42  }   {:a-vector 'a  } {:a-vector :a }
   {:a-vector "a" }   {:a-vector \a  } {:a-vector #inst "2012Z"}
   {:a-vector #{} }   {:a-vector nil } {:a-vector true }
   {:a-vector #{42} }
   {:a-vector false } {:a-vector {:a-vector 'foo} } ) )
```

We cannot just *map* or iterate *get-a-vector* over bad inputs because Clojure evaluates arguments eagerly.[12] The first exception will terminate the entire *map* operation, but we want to test that they all throw exceptions.

One way to defeat applicative-order evaluation is with a higher-order function. [13] We can pass *get-a-vector* as a function to another function that wraps it in a *try* that converts an exception into a string. We then collect all bad-ish strings into a hash-set and test that the hash-set contain just the string "*java.lang.IllegalArgumentException.*" For the *good-ish* test set, we map the values into a sequence that should match the inputs in order.

Finally, we test the good nested cases – r-vectors containing a-vectors – explicitly, without functional tricks.

[12] so-called *applicative-order evaluation*; see `http://en.wikipedia.org/wiki/Applicative_order#Applicative_order`

[13] another, more complicated way is with a *macro*, which rewrites expressions at compile time. Macrros should be avoided when functional alternatives exist because they are hard to develop and debug.

```
(defn- exception-to-name [fun expr]
  (try (fun expr)
       (catch Exception e (re-find #"[^:]+" (str e)))))

(defn- value-seq [fun exprs]
  (map (fn [x] (exception-to-name fun x)) exprs))

(defn- value-set [fun exprs]
  (set (value-seq fun exprs)))

(deftest get-a-vector-helper-test
  (testing "get-a-vector-helper"
    ;; Negative tests
    (is (= #{"java.lang.IllegalArgumentException"}
           (value-set get-a-vector bad-ish-test-set)))
    ;; Positive tests
    (is (= good-ish-test-set
           (value-seq get-a-vector good-ish-test-set)))
    (are [x y] (= x y)
        [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 {}})
    ) ) )
```

4.5 Dispatching Operations by Collection Type

To implement the protocol, we need multimethods that dispatch on the collection types of the a-vectors. Lists and Clojure vectors should be treated the same: as sequences. Let's call them *seq-ish*. Hash-maps should be treated as *map-ish*. All other types are illegal.

```
(defn one-type [a]
  (cond
    (or (vector? a) (list? a)) 'seq-ish
    (map? a)                   'map-ish
    :default (throw (IllegalArgumentException. (str ": " a)))))
```

Test *one-type* on the *bad-ish* set as before, expecting one *map-ish* result because *one-type* is not sensitive to nested r-vectors, by design. Test *one-type* on the *good-ish* set by comparing transformed good values against good inputs. This approach requires us to maintain a parallel sequence of good-ish results, but it's still preferable to writing repetitive code. In general, repetitive data is better than repetitive code.

```
(def ^:private good-ish-collection-types
  '(seq-ish seq-ish map-ish,
    seq-ish seq-ish map-ish,
    seq-ish seq-ish map-ish,
    seq-ish seq-ish seq-ish,
    seq-ish seq-ish seq-ish,
    map-ish map-ish map-ish))
(deftest one-type-test
  (testing "type-merging"
    (is (= #{"java.lang.IllegalArgumentException" 'map-ish}
           (value-set one-type bad-ish-test-set)))
    (is (= good-ish-collection-types
           (value-seq one-type good-ish-test-set))) ) )
```

To dispatch on collection type, we must test the types of all inputs. Here, again, we see ampersands before a parameter representing a sequence of all optional arguments.

```
(defn all-types [& exprs] (set (map one-type exprs)))
(defmulti  add-a-vectors all-types)
(defmethod add-a-vectors #{'seq-ish} [& those]
 (apply map + those))
(defmethod add-a-vectors #{'map-ish} [& those]
 (apply merge-with + those))
(defmethod add-a-vectors :default    [& those]
  (throw (IllegalArgumentException.
    (str "add-a-vectors doesn't like this food: " those))))
```

At this point, it is worth noting that *static typing* – types tested by the compiler – would save us the work of writing run-time tests of these type tests. A statically typed language like Scala or Haskell would save us this work, but at the expense of the build-time and run-time complexity of introducing another language into our data-processing pipeline. This complexity tradeoff – coding versus building and running – is a judgment call, and we stick with dynamic type-checking, the only option available in Clojure, for now.

Our *add-a-vectors* function is quite loose: it will add one or more a-vectors, where our protocol will only accept two or more. This is fine: it only means that we unit test a few more cases for *add-a-vectors* than for our protocol.

For our negative tests, we can only consider the first ten of the bad-ish test set because the later bad-ish cases include hash-maps we should not test here. So we give the lie to our earlier assertion that we need only add cases to our test sets to augment all our tests. However, it's still convenient to use them. From this point on, we write explicit test code rather than data-driven tests to save the complexity of building more test infrastructure. The results are quite lengthy, but repetitive and easy to understand.

For our positive tests, we cover all combinations of list and vector, plus some cases of dimension mismatch. We expect *add-a-vectors* to produce results for the minimum dimension of its inputs. Again, this looseness is by-design at this level of the overall solution.

```
(deftest add-a-vectors-test
  (testing "add-a-vectors")
  (is (= #{"java.lang.IllegalArgumentException"}
         (value-set add-a-vectors
         (take 10 bad-ish-test-set))))
  (is (thrown? java.lang.IllegalArgumentException
       (add-a-vectors)))
  (are [x y] (= x y)
    (add-a-vectors [])    []
    (add-a-vectors [1])   [1]
    (add-a-vectors [1 1]) [1 1]

    (add-a-vectors '())    '()
    (add-a-vectors '(1))   '(1)
    (add-a-vectors '(1 1)) '(1 1)

    (add-a-vectors [1]   [2])   [3]
    (add-a-vectors [1 2] [3 4]) [4 6]

    (add-a-vectors '(1)   '(2))   '(3)
    (add-a-vectors '(1 2) '(3 4)) '(4 6)

    (add-a-vectors '(1)   [2])   [3]
    (add-a-vectors '(1 2) [3 4]) [4 6]

    (add-a-vectors [1]   '(2))   [3]
    (add-a-vectors [1 2] '(3 4)) [4 6]

    (add-a-vectors [1]   [2])   '(3)
    (add-a-vectors [1 2] [3 4]) '(4 6)

    (add-a-vectors '(1)   [2])   '(3)
    (add-a-vectors '(1 2) [3 4]) '(4 6)

    (add-a-vectors [1]   '(2))   '(3)
    (add-a-vectors [1 2] '(3 4)) '(4 6)

    (add-a-vectors [1]   [2 3])   [3]
    (add-a-vectors [1 2] [3 4 5]) [4 6]

    (add-a-vectors {})          {}
    (add-a-vectors {:a 1})      {:a 1}
    (add-a-vectors {:a 1 :b 2}) {:a 1 :b 2}

    (add-a-vectors {:a 1})      {:a 1}
    (add-a-vectors {:a 1 :b 2}) {:a 1 :b 2}
  ) )
```

```
(defrecord ReversibleVector [a-vector]
  IReversibleAlgebraicVector
  (add   [a b & more]
         {:priors 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.