Reversible Arithmetic on Collections

The Team of Fu

September 23, 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

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

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 arithmetic 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 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. Unfortunately, protocol functions don't support "rest" arguments,[10] so we'll go with binary *add* for now, and built up multiarity through functional programming mojo.

Sub should be binary, because multiary sub is ambiguous.

Include inner product, because it is likely to be useful.

[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://bit.ly/18kecbJ`

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

Remark The choice to include operations in a library in the absense of scenarios is philosophical,[11] perhaps more akin to *Action-Centric* design or *proactive* design as opposed to *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 require 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. Unlike object-oriented programming, there is no implicit *this* instance. Here is our protocol design:

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

[11] http://en.wikipedia.org/wiki/Design_philosophy

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*. Therefore, they must be Clojure vectors, lists, or hash-maps.

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

Definition 4.1 (Algebraic Vector (a-vector)) *An **a-vector** is a Clojure vector, a list, or a hash-map that does not contain an* `:a-vector` *attribute.*

Definition 4.2 (Reversible Algebraic Vector (r-vector)) *A **reversible algebraic vector** or **r-vector** is a hash-map containing an* `: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*. 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 can't hold a-vector data: "
             (type that))))))
```

4.3 Fetching *a-vector* Data

We need a way to get a-vector data out of any r-vector.

```
(defn get-a-vector [that]
  (if (not (map? that))
      (throw (IllegalArgumentException. (str that)))
      (check-a-vector (:a-vector 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 test sets for data that should be accepted and rejected immediately. Creating new tests is as easy as adding new instances to these sets. Include some types that may not be acceptable for arithmetic; we are just testing structure here.

```
(def ^:private atoms
  '(42 42.0 42.0M 42N 'a :a "a" \a
    #inst "2012Z"
    #{} #{0} nil true false))

(def ^:private vectors
  (concat [[]] (map vector atoms)))

(def ^:private lists
  (concat [()] (map list atoms)))

(def ^:private maps
  (concat [{}] (map (fn [a] {:a a}) atoms)))

(def ^:private a-vectors
  (concat (map (fn [a] {:a-vector a}) vectors)
          (map (fn [a] {:a-vector a}) lists)
          (map (fn [a] {:a-vector a}) maps)))

(def ^:private good-ish-test-collection
  a-vectors)

(def ^:private bad-ish-test-collection
  (concat maps
          (map (fn [a] {:a-vector a}) atoms)
          (map (fn [a] {:a-vector a}) a-vectors)))
```

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 eager evaluation is with a higher-order function.[13] Pass *get-a-vector* as a function to another function that wraps it in a *try* that converts an exception into a string. Collect all bad-ish strings into a hash-set and test that the hash-set contain only the string "*java.lang.IllegalArgumentException.*" For the *good-ish* test set, map the

[12] http://en.wikipedia.org/wiki/Evaluation_strategy#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.

values into a sequence that should match the inputs in order.

```
(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-test
  (testing "get-a-vector"
    ;; Negative tests
    (is (= #{"java.lang.IllegalArgumentException"}
           (value-set get-a-vector bad-ish-test-collection)))
    ;; Positive tests
    (is (= (map :a-vector good-ish-test-collection)
           (value-seq get-a-vector good-ish-test-collection))) ) )
```

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)))))
```

To dispatch on collection type, test the types of all inputs. Here we see ampersands before a parameter that represents 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 "; Illegal type combination: " (map type those)))))
```

At this point, it is worth noting that *static typing* – types tested by a compiler – would save us the work of writing run-time type tests, 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. We stick with dynamic type-checking, the only option available in Clojure, for now.

Our *add-a-vectors* function is 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.

Regarding the underlying arithmetic: if we attempt to add values that cannot be added via the + operator, we do not interfere with the underlying exceptions that Clojure and Java may throw. Therefore, we do not need to test such cases here.

```
(deftest add-a-vectors-test
  (testing "add-a-vectors")
  (is (= #{"java.lang.IllegalArgumentException"}
         (value-set add-a-vectors atoms)))
  (are [expr] (thrown? java.lang.IllegalArgumentException expr)
       (add-a-vectors '()  {})
       (add-a-vectors  []  {})
       (add-a-vectors  {} '())
       (add-a-vectors  {}  [])
       (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}
```

4.6 The ReversibleVector Type

We now have enough to implement the *add* method of the protocol.

```
(defrecord ReversibleVector [my-r-vector]
  IReversibleAlgebraicVector
  (add   [this that]
         (let [prior-a-vectors
               [(get-a-vector (.my-r-vector this))
                (get-a-vector (.my-r-vector that))]]
           (->ReversibleVector
             {:priors    [this that]
              :operation 'add
              :a-vector  (apply add-a-vectors prior-a-vectors)})))
  (sub   [this that] nil)
  (inner [this that] nil)
  (scale [this scalar] nil)
  (undo  [this] nil)
  (redo  [this] 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.