

# Reversible Arithmetic on Collections

The Team of Fu

September 21, 2013

## Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>A Protocol for Reversible Arithmetic</b>	<b>3</b>
2.1	Implementing the Protocol for Vectors . . . . .	3
<b>3</b>	<b>Unit-Tests</b>	<b>7</b>
<b>4</b>	<b>REPLing</b>	<b>7</b>

## 1 Introduction

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

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

We often encounter data records as hash-maps, sequences, or vectors. Arithmetic on vectors is familiar from school math: to get the sum of two vectors, just add the corresponding elements, first-to-first, second-to-second, and so on. Here’s an example in two dimensions:

We don’t need to write the commas (but we can if we want – in Clojure, they’re just whitespace):

`[1 2] + [3 4] = [4 6]`

Clojure’s *map* can already do this:

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

---

<sup>1</sup>[http://en.wikipedia.org/wiki/Literate\\_programming](http://en.wikipedia.org/wiki/Literate_programming).

==> [4 6]

The same idea works in any number of dimensions and with any kind of elements that can be added (any *field*:<sup>2</sup> integers, complex numbers, quaternions. It's the foundation of the important theory of *Vector Spaces* in mathematics.<sup>3</sup>

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

```
...
[-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]
...
```

and a large infinity of more answers.

But, in our financial computations, we usually want this functionality so we can undo a mistake, roll back a provisional result, perform a backfill or allocation: in short, get back the original inputs.

Let's define a protocol for *reversible arithmetic in vector spaces* that captures the desired functionality. We want a protocol because we want several implementations with the same reversible arithmetic. For instance, we should be able to do similarly for hash-maps, which, after all, are just sparse vectors with named components:

```
{:x 1, :y 2} + {:x 3, :y} = {:x 4, :y 6}
```

To get the desired behavior, we can't use *map*; it doesn't work the same on hash-maps. We must use Clojure's *merge-with*:

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

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

We want to get rid of these annoying differences: the protocol for reversible arithmetic on data rows should be the same for all collection types.<sup>4</sup> Along the way, we'll do some hardening so that the implementations are robust both mathematically and computationally.

---

<sup>2</sup>[http://en.wikipedia.org/wiki/Field\\_\(mathematics\)](http://en.wikipedia.org/wiki/Field_(mathematics))

<sup>3</sup>[http://en.wikipedia.org/wiki/Vector\\_space](http://en.wikipedia.org/wiki/Vector_space)

<sup>4</sup>including streams over time! Don't forget Rx and SRS.

## 2 A Protocol for Reversible Arithmetic

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. Don’t misread *IReversibleAlgebraicVector* as “irreversible algebraic vector,” but rather read it as “Interface to Reversible Algebraic Vector,” where “Interface” is a synonym for “protocol.”

```
(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])
)
```

### 2.1 Implementing the Protocol for Vectors

As a first cut, package 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 vectors. What kinds of things can hold ordinary vector data? They be things we can operate on with *map* or *merge-with* to perform the basic, vector-space operations. Therefore, the data must be a Clojure vector, list, or hash-map.

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 2.1 (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 al-

*gebraic vector does contain a `:data` attribute, the value of that attribute is a base-data collection.*

Here is a *fluent*, type-checking function that 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
            (= t (type {})))
        that
        (throw (IllegalArgumentException.
                (str "This type of object can't hold vector data: " t))))))
```

Now we can formally define the reversible algebraic vector:

Now we need a way to get the data out of any reversible algebraic vector. 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* and a hash-map that has no `:data`; the prettier code

```
(:data that)
```

does not distinguish those two cases.

```
(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))))
```

```

(deftest get-data-helper-test
  (testing "get-data-helper"
    (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 })
    )
    (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 {}})
    )
  ))

```

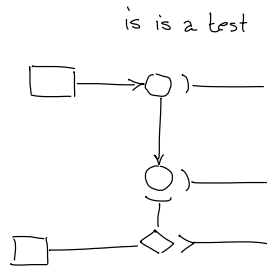


Figure 1: This means “Fortune” and is pronounced “Fu”.

Here are unit tests for these helpers that show how they enforce the definition.

```
(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))
```

### 3 Unit-Tests

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

### 4 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.