Reversible Arithmetic on Collections

The Team of Fu

September 20, 2013

Contents

1 Introduction

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

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:

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

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]
```

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

Clojure's *map* can already do this:

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

==> [4 6]
```

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

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}
```

[2] `http://en.wikipedia.org/wiki/Field_(mathematics)`
[3] `http://en.wikipedia.org/wiki/Vector_space`

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

2 A Protocol for Reversible Arithmetic

First, 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 Clojure vectors in hash-maps that contain enough information to reverse any computation.

Start with a little helper to get data from a vector that may be either a basic Clojure vector or one of our reversible algebraic vectors. Define it as a *multimethod* since we anticipate needing it for every basic type that can hold a reversible algebraic vector, namely for vectors, lists, and hash-maps. If the hash-map has a value for key *:data*, then return that value. Otherwise, return the given hash-map. This implies our first rule:

[4] including streams over time! Don't forget Rx and SRS.

Rule 2.1 (Data) *A reversible algebraic vector is a hash-map, list, or ordinary Clojure vector. If it is a hash-map, it either has a* `:data` *attribute or not. If it has a* `:data` *attribute, then its data is the value of that attribute. Otherwise, its data is the hash-map is itself.*

```
(defmulti get-data type)
(defmethod get-data (type [])  [that] that)
(defmethod get-data (type '()) [that] that)
(defmethod get-data (type {}) (or (:data that) 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)
  (undo [b] nil))
```

3 Functions

```
(def x 42)
(defn foo [] x)
```

4 Unit-Tests

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

```
(deftest null-test
  (testing "null test"
    (is (= (merge-with + {:x 1 :y 2} {:x 3 :y 4}) {:x 4 :y 6}))
    (is (= 1 1))
  ))
```

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.

```
(run-all-tests)
```

```
(foo)
```