Fluent, Composable Error Handling

Brian Beckman

September 8, 2013

Contents

1 Introduction

Consider a program composed of *serially dependent computations*, any of which produces either a value to feed to the next computation in line or an error. If any computation in the sequence produces an error, no downstream computations should be attempted and that error should be the result of the entire sequence.

We show how to write such a program in fluent style, which minimizes the number of temporary variables that must be invented and named. This style directly mimics the abstract data flow of the solution. We further show techniques for fluent error handling, both with exceptions and with returned error codes.

2 Motivating Example Problem

As a concrete example, suppose we must get an authorization token, do a database lookup, do a web-service call, filter the results of that call, do another web-servie call, and then combine the results. The data flow of our program resembles that in figure 1.

In the C++ – like languages, including JavaScript and Java, we might keep intermediate results in instance variables and model the flow as methods that produce instances from instances, that is, as **transforms**, also called **fluent style**. Imagine a *main* program like the following, in which put transforms on their own lines, indented from their *sources* by one 4-space tab stop. The **source** of a transform is an expression that produces an object.

```
public static void main(String[] args) {
    Computation databaseResults = new Computation()
        .authorize()
        .readDatabase();
    String result = databaseResults
        .callWebService()
        .filterResults()
        .combineResults(databaseResults
            .callOtherWebService());
    System.out.println(result); } }
```

Notice that we save the *Computation* produced by reading the database in its own local variable, namely *databaseResults.* We do so because the dataflow branches from that result and we need to use it twice, once as the source for calling the first web service and once as the source for calling the second web service. If not for this branching and recombining of the dataflow, we might have written the entire program as one, fluent expression.

```
public class Computation {
    private String authToken;
    private String databaseResults;
    private String webServiceCallResults;
    private String filteredWebServiceCallResults;
    private String otherWebServiceCallResults;

    public Computation () {}
    public Computation authorize() {
        authToken = "John's credentials";
        return this; }
    public Computation readDatabase() {
```

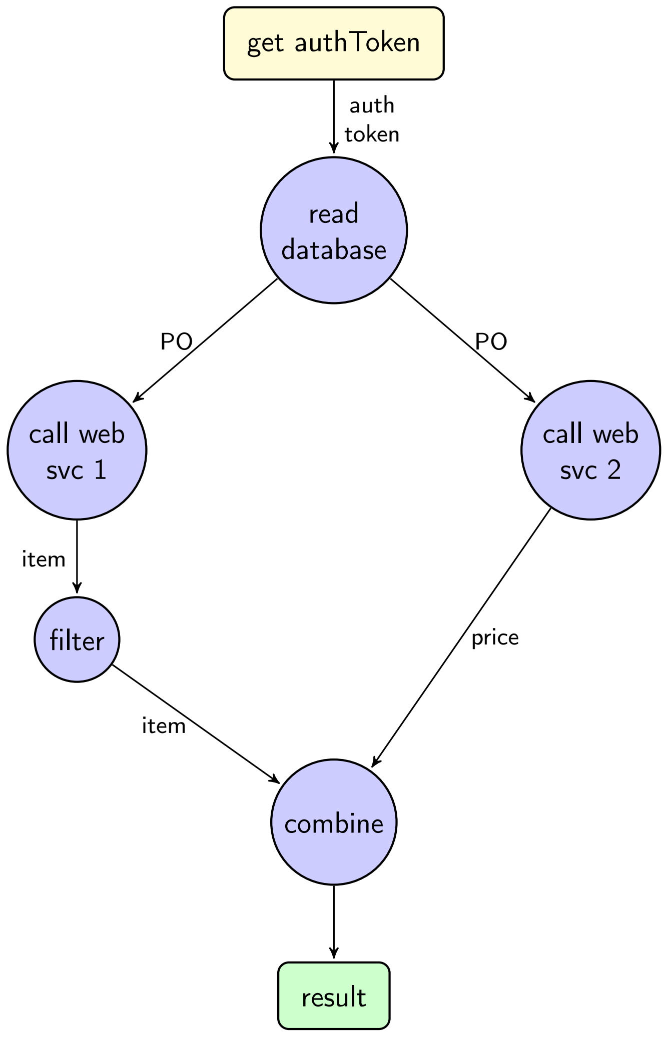

Figure 1: Serially dependent computations

```
        databaseResults = "{\"name\":\"John\", \"PO\":\"421357\"}";
        return this; }
    public Computation callWebService() {
        webServiceCallResults =
            "{\"item\":\"camera\", \"item\":\"shoes\"}";
        return this; }
    public Computation filterResults() {
        filteredWebServiceCallResults =
            "{\"item\":\"camera\"}";
        return this; }
    public Computation callOtherWebService() {
        otherWebServiceCallResults = "{\"price\":\"420.00\"}";
        return this; }
    public String combineResults(Computation other) {
        return "{[" + filteredWebServiceCallResults +
            "," + otherWebServiceCallResults + "]}"; }

    public static void main(String[] args) {
        Computation databaseResults = new Computation()
            .authorize()
            .readDatabase();
        String result = databaseResults
            .callWebService()
            .filterResults()
            .combineResults(databaseResults
                .callOtherWebService());
        System.out.println(result); } }
```

One of the better ways to write such a program is to package every computation in a function (or method) that either produces a correct value or throws an exception.

So packaged, we may write the program directly as a sequence *via* Clojure's `->` or `->>` or the `let` syntax, as follows:

```
(try
  (let [auth-token        (get-auth-token)
        db-results        (read-database auth-token)
        svc-results       (call-web-service db-results)
        other-svc-results (call-other-web-service svc-results)
        filtered-results  (filter my-predicate
    read-database

  (catch Exception e (.getMessage e)))
```

The desired behavior is similar to that of the Maybe monad,[1] the difference being that *Maybe* just produce *Nothing* if anything goes wrong. The consumer of the computation doesn't know what stage of the pipeline failed nor any details at all about the error. *Maybe* suppresses all that. Such a situation is not tolerable in the real world. Consider the example of a database retrieval followed by a few web-service calls followed by a filter and transformation followed by a logging call followed by output to UI components. If something goes wrong in this sequence of computations, we need to know exactly where and as much detail as we can get about the failure. But we certainly don't want any computations downstream of the failure to be attempted.

3 Code

```
(defproject ex1 "0.1.0-SNAPSHOT"
  :description "Project Fortune's Excel Processor"
  :url "http://example.com/TODO"
  :license {:name "TODO"
            :url "TODO"}
  :dependencies [[org.clojure/clojure     "1.5.1"]
                 [org.clojure/algo.monads "0.1.4"]
                 [org.clojure/data.zip    "0.1.1"]
                 [dk.ative/docjure        "1.6.0"]
                ]
  :repl-options {:init-ns ex1.core})



(ns ex1.core
  (:use clojure.algo.monads))
```

[1] http://en.wikipedia.org/wiki/Monad_(functional_programming)#The$_{\text{Maybe monad}}$

```
(defmonad if-not-error-m
  [m-result (fn [value] value)
   m-bind   (fn [value f]
              (if-not (:error value)
                (f value)
                value))
   m-zero   {:error "unspecified error"}
   m-plus   (fn [& mvs]
              (first (drop-while :error mvs)))

   ])
```

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

```
(deftest exception-throwing-test
  (testing "exceptions are thrown"
    (is (thrown? ArithmeticException (/ 1 0)))
    (is (thrown-with-msg? ArithmeticException #"Divide by zero" (/ 1 0)))
    ))

(deftest comprehension-test
  (testing "sequence monad and comprehension"
    (is (= (domonad sequence-m
                    [a (range 5)
                     b (range a)]
                    (* a b))
           (for [a (range 5)
                 b (range a)]
             (* a b)))
        "Monadic sequence equals for comprehension")))

(defn- divisible? [n k]
  (= 0 (rem n k)))

(def ^:private not-divisible?
  (complement divisible?))

(defn- divide-out [n k]
  (if (divisible? n k)
    (recur (quot n k) k)
    n))

(defn- error-returning-check-divisibility-by [k n]
  (let [q (divide-out n k)]
    (if (= q n)
      {:error (str n ": not divisible by " k)}
      q)))

(defn- exception-throwing-check-divisibility-by [k n]
  (let [q (divide-out n k)]
    (if (= q n)
      (throw (Exception.
              (str {:error (str n ": not divisible by " k)})))
      q)))

(defn- best-small-divisor-sample [a2]
  (try
    (->> a2
        (exception-throwing-check-divisibility-by 2)
        (exception-throwing-check-divisibility-by 3)
        (exception-throwing-check-divisibility-by 5)
        (exception-throwing-check-divisibility-by 7))
    (catch Exception e (.getMessage e)))
```

4 References

5 Conclusion