Dynamic Data Pipelines

Robert S. Smith*

April 26, 2013

Abstract

Communication between different points in a call stack is often done by setting up an explicit data pipeline between the two points. In this paper, we present a method using Common Lisp's condition system in order to eliminate an explicit pipeline. We describe this by studying a generalizable problem.

1 The Common Lisp Condition System

The Common Lisp condition system is a powerful system principally designed for flexible handling of program conditions (most often program errors). The sytem operates similarly to other languages' exception handling systems, except for two main differences.

Firstly, conditions can be handled interactively by the user by invoking the *interactive debugger*. Typically, when an error occurs, the operator may choose from a list of options which act as recourse for the error.

Secondly, and more importantly, program conditions are *restartable*. In short, if a condition is *signalled* at a particular stack frame, Lisp will walk up the call stack until the condition is *handled*. At the option of the handler, a *restart* maybe be *invoked*, which will cause Lisp to *walk back down the stack* to the original point of the error, and perform particular actions prescribed by the restart.

For example, suppose we have a list of points (represented as cons cells) and we wish to compute the slope between successive points.

```
(defun slope (p q)
  ;; Division can signal the error DIVISION-BY-ZERO.
  (/ (- (cdr p) (cdr q))
     (- (car p) (car q))))
```

Now, we write a function to compute the slopes, but we provide different ways to "restart the computation" in the event our `slope` function divides by zero.

*E-mail address: `quad@symbo1ics.com`

```
(defun compute-slopes (points)
  (loop :for p :in points
        :for q :in (cdr points)
        :collect (restart-case (slope p q)
                   (return-nil ()
                     :report "Return␣NIL."
                     nil)
                   (return-zero ()
                     :report "Return␣zero."
                     0)
                   (specify-value (value)
                     :report "Specify␣a␣value␣to␣return."
                     :interactive (lambda ()
                                    (format t "Enter␣a␣value:␣")
                                    (list (read)))
                     value))))
```

Here, we provide three different ways for the user to restart. If `slope` divides by zero, we can...

1. return `nil`,
2. return `0`, or
3. allow the user to specify a value at runtime to use.

If we call `compute-slopes` with erroneous data, the following happens[1]:

```
> (compute-slopes '((4 . 1) (4 . 3) (9 . 2)))

  arithmetic error DIVISION-BY-ZERO signalled
Operation was SB-KERNEL::DIVISION, operands (-2 0).
   [Condition of type DIVISION-BY-ZERO]

Restarts:
 0: [RETURN-NIL] Return NIL.
 1: [RETURN-ZERO] Return zero.
 2: [SPECIFY-VALUE] Specify a value to return.
```

If we choose option 0, we get `(NIL -1/5)` as a result.

Suppose the above is library code, and for another application, we are writing code to compute angles between points. Given a slope m, we can compute the angle the slope represents in degrees with $\frac{180}{\pi}\tan^{-1} m$.

```
(defun angle (m)
  (round (* (/ 180 pi) (atan m))))
```

To compute the angles of the points, we just apply this function to successive slopes:

```
(defun compute-angles (points)
  (mapcar #'angle (compute-slopes points)))
```

[1]Implementations often provide other options as well.

If we run into a division-by-zero error, we still have to invoke the restart manually. We can automate this by handling the error and invoking a restart with `invoke-restart`. For slopes computations that divide by zero, it is sensible to just return a very high value, since a large slope indicates vertically colinear points. Our `compute-angle` function becomes

```
(defun compute-angles (points)
  (handler-bind
      ((division-by-zero
         (lambda (c)
           (declare (ignore c))
           (invoke-restart 'specify-value 150))))
    (mapcar #'angle (compute-slopes points))))
```

Now if we call `compute-angles` on the “erroneous” data, we do not actually get an error:

```
> (compute-angles '((4 . 1) (4 . 3) (9 . 2)))
(90 -11)
```

Indeed, 90° is the angle between two vertical points relative to the horizontal.

2 A Pipelining Problem

Suppose we wish to write a library to lint[2] C programs. The linter would be structured as in the following Common Lisp function:

```
(defun lint (file)
  (let ((ast (parse file)))
    (type-check ast)
    (arity-check ast)))
```

That is, parse the C file into an abstract syntax tree, then perform our checks (in this case, a type check and a check to see all function calls have the right arity).

How are warnings and errors reported to the user? The most naïve approach would be to simply print out the warnings and errors. Our `type-check` function might look like:

```
(defun type-check (ast)
  ;; ...
  (format t "WARNING:␣Type␣mismatch.")
  ;; ...
  )
```

During parsing, we might run into improper syntax from which we cannot recover (this is called a *fatal condition*). If such an error is encountered, it would not be desirable to continue with the linting process. We can print the condition, call `error`, and abort the computation.

[2] *Linting* is a form of static analysis which attempts to find common programmer errors.

Now suppose we wish to encapsulate our lint conditions in some kind of object so we can include auxiliary information, such as source location information. Aside from packaging up extra data, having the lint conditions encapsulated in a data object would allow us to programmatically use it[3].

To define the objects, we just define a few structures.

```
(defstruct lint-condition source-location message)

(macrolet ((defcondition (name)
             `(defstruct (,name (:include lint-condition))
                ;; empty
                )))
  (defcondition lint-advice)
  (defcondition lint-warning)
  (defcondition lint-error)
  (defcondition lint-fatal))
```

Also suppose we will want to process these conditions in some way. Perhaps we will want to display the errors in order of increasing severity to the user.

```
(defun severity-sort (conditions)
  (flet ((severity-to-int (x)
           (etypecase x
             (lint-advice  0)
             (lint-warning 1)
             (lint-error   2)
             (lint-fatal   3))))
    (sort (copy-list conditions) #'< :key #'severity-to-int)))
```

Instead of reporting the conditions as they are encountered, we will collect them. One option would be to return a list of encountered conditions from each of the analysis functions and concatenate them together, but this doesn't work so well when functions have a natural alternative return type, like `parse`. As such, we might pass an extra argument to each function which contains the collected conditions. To do this, we can create a mutable cell in the spirit of Standard ML's `ref`:

```
(defstruct ref contents)
```

Now our linter will look like this:

```
(defun lint (file)
  (let* ((conds (make-ref))
         (ast (parse file conds)))
    (type-check ast conds)
    (arity-check ast conds)

    (report-conditions conds)))

(defun type-check (ast conditions)
```

[3]For example, we might want to visually display the issues in the operator's editor.

```
  ;; ...
  (push (make-lint-warning :source-location ...
                           :message "Type␣mismatch.")
        (ref-contents conditions))
  ;; ...
  )

(defun report-conditions (conditions)
  (dolist (c (severity-sort (ref-contents conditions)))
    (format t "At␣~A:␣~A~%"
            (lint-condition-source-location c)
            (lint-condition-message c))))
```

Unfortunately, if we reach a fatal condition, the conditions won't be reported. With a slight modification, we have the following definition of `lint` with an example use of reporting a fatal condition:

```
(defun lint (file)
  (let ((conds (make-ref)))
    (handler-bind ((error (lambda (e)
                            (declare (ignore e))
                            (report-conditions conds))))
      (let ((ast (parse file conds)))
        (type-check ast conds)
        (arity-check ast conds)

        (report-conditions conds)))))

(defun parse (file conditions)
  ;; ...
  (push (make-lint-fatal :source-location ...
                         :message "Missing␣brace.")
        (ref-contents conditions))
  (error "Abort␣linting.")
  ;; ...
  )
```

This method of reporting conditions works relatively well until the program becomes more complex. For example, `check-arity` might call another function called `check-node`. Each `check-*` function essentially has to keep track of conditions explicitly to maintain a data pipeline. Additionally, we duplicate the sites where conditions are reported in the `lint` function. Our program quickly becomes encumbered in bookkeeping code which is largely irrelevant to the actual functionality.

3 Removing the Explicit Pipeline

The problem can be boiled down to the need to talk between two different points in the call stack. Before we lint, it would be ideal if we could set up

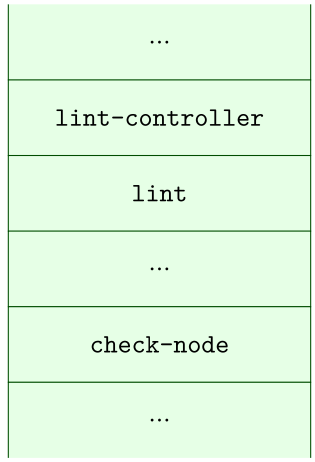

Figure 1: Diagrammatic representation of a downward-growing call stack.

a sort of functionality that could receive the lint conditions as they are generated, and optionally take over control in the event a fatal condition occurs. We would have a `lint-controller` function which controls the collection of conditions and execution of the linting process. And then from any of the functions called within `lint-controller` (i.e., at any stack depth below that of the call to `lint-controller`, as depicted in figure 1), we could send any new program condition back up for collection, and optionally tell the controller to not resume control.

This suggests two interfaces: a *sender* and *receiver*. The receiver establishes a context with which the sender can communicate.

In order to communicate, we need to be able to traverse the call stack at any time in the program without destroying it. As seen in §1, signalling conditions lets us traverse the stack upwards, and invoking restarts lets us traverse the stack downwards. Typically, conditions carry around information relevant to the state of the program (e.g., function arguments that caused the error), however, they can carry any data freely. This motivates the definition of a "messenger" condition:

```
(define-condition messenger (condition)
  ((payload :initarg :payload
            :reader messenger-payload)))
```

To send the condition up the stack, we simply signal it as an error.

```
(defun collect (data)
  (error 'messenger :payload data))
```

However, we need to be able to restart the computation, so we wrap the error in a restart.

```
(defun collect (data)
```

```
  (restart-case (error 'messenger :payload data)
    (continue ()
      :test (typep condition 'messenger)
      nil)))
```

We must put the test there to ensure we can only handle the `messenger` conditions (i.e., we don't want to be able to restart any other error conditions here).

In the lint controller, we want to intercept the message, and invoke a restart. However, before invoking the restart, we push the payload of the messenger onto our messages list.

```
(defun lint-controller (file)
  (let ((messages nil))
    (handler-bind
        ((messenger (lambda (m)
                      (push (messenger-payload m) messages)
                      (invoke-restart
                       (find-restart 'continue m)))))
      (lint file))
    (process-messages (nreverse messages))))
```

By calling `collect` within the dynamic scope of the handler, we can collect any data we want. However, there are a few problems we will solve in the next sections:

1. Fatal conditions will not abort the collection process.

2. The collecting logic is not sufficiently abstract for reusability.

3. Collections aren't composable.

Allowing For Fatal Conditions

In the event of a fatal condition, we want to abort collection immediately. In order to do this, we modify our `collect` function to carry a flag on whether or not to continue computation. We also must enrich our `messenger` condition to carry this data.

```
(define-condition messenger (condition)
  ((payload :initarg :payload
            :reader messenger-payload)
   (continuep :initarg :continuep
              :initform t
              :reader messenger-continuep)))

(defun collect (data &key (continuep t))
  (restart-case (error 'messenger :payload data)
    (continue ()
      :test (typep condition 'messenger)
      nil)))
```

Finally, we add new logic to the lint controller.

```
(defun lint-controller (file)
  (let ((messages nil))
    (block collector
      (handler-bind
          ((messenger (lambda (m)
                        (push (messenger-payload m) messages)
                        (if (messenger-continuep m)
                            (invoke-restart
                             (find-restart 'continue m))
                            (return-from collector
                              (process-messages
                               (nreverse messages)))))))
        (lint file))
      (process-messages (nreverse messages)))))
```

Packaging Up the Logic

The function `lint-controller` is showing a common pattern in Lisp of setting up state and doing some actions. This is a typical use of a `with-*` style macro.

The strategy is to package up the lint controlling logic into a macro called `with-dynamic-collection`, which returns the list of messages we received. This will reduce `lint-controller` to the following:

```
(defun lint-controller (file)
  (process-messages
   (with-dynamic-collection ()
     (lint file))))
```

The extra empty list passed to `with-dynamic-collection` will become useful later.

Packaging up into a macro is relatively straightforward. We simply abstract out the body of the computation and give unique names to any symbols that we don't want to introduce lexically to the body.

```
(defmacro with-dynamic-collection (() &body body)
  (let ((messages (gensym "MESSAGES-"))
        (block-name (gensym "BLOCK-NAME-")))
    `(let ((,messages nil))
       (block ,block-name
         (handler-bind
             ((messenger (lambda (m)
                           (push (messenger-payload m) ,messages)
                           (if (messenger-continuep m)
                               (invoke-restart
                                (find-restart 'continue m))
                               (return-from ,block-name
                                 (nreverse ,messages))))))
           ,@body)
         (nreverse ,messages)))))
```

Making Dynamic Collection Composable

Consider the following code:

```
(with-dynamic-collection ()              ; A
  (collect 1)
  (with-dynamic-collection ()            ; B
    (collect 2)
    (collect 3)))
```

Here, the collector marked “A” will return the list `(1)` while the collector marked “B” will return `(2 3)`. Clearly, when we have nested collectors, all `collect` calls will send data only to the innermost collector, even across function call boundaries. Ideally, we could collect values at any level of the program to any collector (provided the `collect` calls are dynamically inside the `with-dynamic-collection` form).

We can do this with the notion of *tags*, similar to the Common Lisp concept of `block` or `tagbody` tags. If we wanted the above collectors to return `(1 3)` and `(2)` respectively, we might have the following:

```
(with-dynamic-collection (:tag :outer)   ; A
  (collect 1 :tag :outer)
  (with-dynamic-collection (:tag :inner) ; B
    (collect 2 :tag :inner)
    (collect 3 :tag :outer)))
```

The strategy for implementing this is simple. We equip the messengers with a tag, so that when we reach a dynamic collection form, we can test whether or not to collect, or to simply pass it off. However, when we invoke a restart, we need to know which `collect` form to return to. As such, when we call `collect`, we also generate a unique identifier which gets attached to the messenger as well, and we add an equality test to the `:test` form of the restart.

First, we add new slots to the messenger.

```
(define-condition messenger (condition)
  ((payload :initarg :payload
            :reader messenger-payload)
   (continuep :initarg :continuep
              :initform t
              :reader messenger-continuep)
   (id        :initarg :id
              :reader messenger-id)
   (tag       :initarg :tag
              :initform nil
              :reader messenger-tag)))
```

Next, we add new logic to the collection form.

```
(defmacro with-dynamic-collection ((&key tag) &body body)
  (let ((messages (gensym "MESSAGES-"))
        (block-name (gensym "BLOCK-NAME-"))
        (tag-once (gensym "TAG-ONCE-")))
```

```
    `(let ((,messages nil)
           (,tag-once ,tag))
       (block ,block-name
         (handler-bind
             ((messenger (lambda (m)
                           (when (eql ,tag-once
                                      (messenger-tag m))
                             (push (messenger-payload m)
                                   ,messages)
                             (if (messenger-continuep m)
                                 (invoke-restart
                                  (find-restart 'continue m))
                                 (return-from ,block-name
                                   (nreverse ,messages)))))))
           ,@body)
         (nreverse ,messages)))))
```

Finally, we add a tag and unique identifier to the `collect` function.

```
(defun collect (data &key (continuep t) tag)
  (let ((id (gensym "ID-")))
    (restart-case (error 'messenger :payload data
                                    :continuep continuep
                                    :id id
                                    :tag tag)
      (continue ()
        :test (lambda (condition)
                (and (typep condition 'messenger)
                     (eq id (messenger-id condition))))
        nil))))
```

Making `Collect` a No-Op

The final change we will make has more to do with code maintenance than the correctness of the dynamic collector.

The `collect` function, as it stands, acts as a no-op at its call site; while it does send data for collection, it just returns `nil`. In our original lint example, while incrementally developing the program, we would run into a lot of errors when calling functions which in turn call `collect`, because no `with-dynamic-collection` context would be set up. It would be advantageous to allow `collect` to not require an outer collecting form, thereby truly making it a no-op.

We can do this by making a runtime-configurable flag which ensures that `collect` is within a `with-dynamic-collection` form.

```
(defvar *ensure-handled-collect* nil)
```

If this flag is `t`, instead of *erroring* inside of our `collect`, we will simply *signal* the condition. This has the effect that if the condition isn't handled, control will simply return back to the `collect` function. In addition to changing the signalling semantics, we will also allow the user to return an arbitrary value from the `collect` form.

```
(defun collect (data &key return (continuep t) tag)
  (let ((id (gensym "ID-")))
    (restart-case (if *ensure-handled-collect*
                      (error 'messenger :payload data
                                        :continuep continuep
                                        :id id
                                        :tag tag)
                      (or (signal 'messenger :payload data
                                             :continuep continuep
                                             :id id
                                             :tag tag)
                          return))
      (continue ()
        :test (lambda (condition)
                (and (typep condition 'messenger)
                     (eq id (messenger-id condition))))
        return))))
```

Finally, if we do indeed signal an error, we want a useful error message. We can do this by adding a report function to our messenger.

```
(define-condition messenger (condition)
  ((payload :initarg :payload
            :reader messenger-payload)
   (continuep :initarg :continuep
              :initform t
              :reader messenger-continuep)
   (id        :initarg :id
              :reader messenger-id)
   (tag       :initarg :tag
              :initform nil
              :reader messenger-tag))
  (:report (lambda (condition stream)
             (declare (ignore condition))
             (format stream
                     "~A was used outside of a ~A form."
                     'collect
                     'with-dynamic-collection))))
```

4 Extensions

The technique described here manifested itself through data collection. However, the technique is relatively general, and can be used for other purposes. `With-dynamic-collect` was very data-oriented, except for the fact one could transfer control back to `collect` by setting a `continuep` flag to `nil`. We could exploit this generally to have a sort of dynamic `goto`: simply return bogus data to a specified tag and set the continue flag to `nil`. We could streamline this by removing all of the logic for actually collecting values.

We could also generalize the above by allowing arbitrary processing of collected values. As it stands, we are just pushing values onto a list. This can be easily and dynamically changed to allow an arbitrary function call. For example, instead of appending to a list, we might create some sort of graphical notification upon receipt of data, or we might insert the value into a more sophisticated data structure, such as a sorted tree.

Appendix A Obtaining and Using the Code

The source code, including documentation, can be obtained from the following url:

`https://bitbucket.org/tarballs_are_good/dynamic-collect/`

The code in this paper and at the aforementioned link is available under the BSD 3-clause license, a copy of which can be found at the above link in the file called "LICENSE".

The code, under the same above license, can also be used via Zach Beane's Quicklisp, via `(ql:quickload :dynamic-collect)`.