Streams-based, Multi-threaded News Classification

European Lisp Symposium 2013
Jason Cornez
CTO, RavenPack

Road to Lisp

- Self-taught - BASIC, Pascal
- MIT 6.001 SICP - Abelson and Sussman
- Silicon Valley - C++, Java, Cold Fusion, PL/SQL
- Move to Europe - Lisp at RavenPack

RavenPack

- Realtime News Analysis Service
- 100,000+ Stories per Day
- 25+ Year News Archive
- Produce News Analytics
- 24/7 Data Service

  Low-Latency, High-Uptime Required

The Challenge

- Multiple Classifiers
- Classifier Dependencies
- Maintainable Code
- Classify each Story in < 200ms

Four Approaches

- Basic Approach
- Threaded Approach
- Synchronous Framework
- Streams Framework

Basic Approach

```
(defclass classifier ()
  ((name :initform “Basic Classifier”)
   (classifier-id :reader classifier-id)
   ...)
  (:documentation “Classifier Base Class”))

(defclass results ()
  ((classifier-id :initarg :classifier-id
                  :reader classifier-id)
   (data          :initarg :data
                  :reader results-data))

(defmethod classify ((classifier classifier) story)
  ;; implement in subclass
  (make-instance ‘results
    :classifier-id (classifier-id classifier)
    :data          ‘not-yet-implemented))
```

Basic Approach

```
(defclass manager ()
  ((name :initarg :manager-name :reader manager-name)
   (source ...) (destination ...)
   (classifiers :reader classifiers)
   ...)
  (:documentation “Classifier Manager”))

(defmethod classify ((manager manager) story)
  (loop
      for classifier in (classifiers manager)
      collect
        (classify classifier story)))
```

Basic Approach

- Pro: Simple interface to classifier
- Con: Single Threaded
- Con: Difficult for one classifier to see results from another

Threaded Approach

```
(defclass classifier ()
  ((name          :initform “Threaded Classifier”)
   (work-queue    :initform (new-queue) :reader work-queue)
   (result-queue :initform (new-queue) :reader result-queue)
   (work-thread  :initform nil :reader work-thread)
   ...))

(defmethod start-classifier ((classifier classifier)) ...)
(defmethod stop-classifier  ((classifier classifier)) ...)

(defmethod classify-loop ((classifier classifier))
  (loop with my-thread = mp:*current-process*
        for story  = (dequeue (work-queue classifier) :wait t)
        for result = (classify classifier story)
        do
          (enqueue (result-queue classifier) result))
        while
          (eq (work-thread classifier) my-thread)))
```

Threaded Approach

```
(defmethod classify ((manager manager) story)
  (loop
      for classifier in (classifiers manager)
      for work-queue = (work-queue classifier)
      do
        (enqueue work-queue story))
 (loop
      for classifier in (classifiers manager)
      for result-queue = (result-queue classifier)
      collect
        (dequeue result-queue :wait t)))
```

Threaded Approach

- Pro: Multi-threaded
- Con: Ordering not possible
- Con: Dependencies not possible
- Con: Each classifier “owns” a thread
- Con: One story at a time

What we Want

```
(define-classifier-manager (:els-13)
    ((:source...)
     (:destination ...)
     (:classifiers
      (:parallel (:serial classifier-A
                          classifier-B)
                 classifier-C
                 classifier-D))))
```

Synchronous Framework

- Make results part of story object
- Introduce way to compose classifiers
- Serial classifier
- Parallel classifier

Results Safely part of Story

```
(defclass story ()
  ((story-id ...) (timestamp ...)
   (headline ...) (body ...)
   (results :initform (make-hash-table)
            :reader story-results)))

(defclass results ()
  ((classifier-id :reader classifier-id)
   (data          :reader results-data))

(defmethod add-results ((story story)
                        (results results))
  (setf (gethash (classifier-id results)
                 (story-results story))
        results))
```

Base Classifier

```
(defclass classifier ()
  ((name :initform "Base Classifier")
   (gate :initform (mp:make-gate t)
         :documentation “Closed when classifying”)
   ...))

(defmethod classify :around ((classifier classifier)
                             (story story))
  (with-slots (gate) classifier
    (mp:close-gate gate)
    (unwind-protect (call-next-method)
      (mp:open-gate gate))))

(defmethod classifier-wait ((classifier classifier))
  (with-slots (gate) classifier
    (mp:process-wait “Waiting for Classification”
                     #'mp:gate-open-p gate)))
```

Serial Classifier

```
(defclass serial-classifier (classifier)
  ((name     :initform "Serial Classifier")
   (children :initform nil :initarg :children
    :documentation "Ordered list of classifiers
                    to run sequentially"))
  (:documentation
   "Classify a story, running each child classifier
    on the story sequentially in the order they
    appear in the list of children."))

(defmethod classify ((classifier serial-classifier)
                     (story story))
  (with-slots (children) classifier
    (dolist (child children)
      (classify child story)))
  story)
```

Parallel Classifier

```
(defclass parallel-classifier (classifier)
  ((name              :initform "Parallel Classifier")
   (children          :initform nil :initarg :children)
   (child-gates       :initform nil
    :documentation "To signal that story is ready")
   (current-story :initform nil
    :documentation "Story to be classified")
   ...)
  (:documentation
   "Classify a story, running each child classifier
    in a separate thread.  The classify method returns
    after all the children have completed."))
```

Parallel Classifier

```
(defmethod startup ((classifier parallel-classifier))
  (with-slots (children child-gates) classifier
    ;; For each child except the first,
    ;; create a closed gate and start a thread
    (loop
       for count from 1 and child in (rest children)
       for gate = (mp:make-gate nil)
       for thread-name =
         (format nil "Parallel child ~d: ~a"
                     count (classifier-name child))
       do
         (push gate child-gates)
         (run-in-thread 'classify-child
            :name thread-name
            :params (list classifier child gate)))))
```

Parallel Classifier

```
(defmethod classify-child
           ((classifier parallel-classifier)
            (child classifier) gate)
  (with-slots (current-story) classifier
    ;; Loop forever, waiting for a signal to process
    ;; each story.
    (loop with wait-reason =
                (format nil “~a Waiting for story”
                                (classifier-name child))
       do (mp:process-wait wait-reason
                           #'mp:gate-open-p gate)
          (mp:close-gate gate)
          (if current-story
              (classify child current-story)
            (loop-finish)))))
```

Parallel Classifier

```
(defmethod classify ((classifier parallel-classifier)
                     story)
  (with-slots (children
               current-story
               child-gates) classifier
    ;; set the current story and signal children
    (setq current-story story)
    (mapc #'mp:open-gate child-gates)
    ;; use current thread to classify the first child
    (classify (first children) current-story)
    ;; wait for all children to complete this story
    (mapc #'classifier-wait (rest children))
    ;; reset current-story
    (setq current-story nil))
  story)
```

Synchronous Framework

- Pro: Simple interface to classifier
- Pro: Full control over dependencies
- Pro: Some multi-threading
- Con: Serial classifier not pipelined
- Con: One story at a time

Streams Framework

- Return to 6.001 / SICP
- Streams allow asynchonous processing
- Allows pipelined serial classifier
- More efficient use of computing resources
- Requires each classifier be a pure function

Streams Review - Scheme

```
(delay exp)         <==> (memo-proc (lambda () exp))
(cons-stream a b) <==> (cons a (delay b))

(define (integers-from n)
  (cons-stream n (integers-from (1+ n))))

(define (sieve stream)
  (cons-stream
   (head stream)
   (seive (filter
           (lambda (x)
             (not (divisible? x (head stream))))
           (tail stream)))))

(define primes (sieve (integers-from 2)))
```

Streams in Common Lisp

```
(defun memo-proc (proc)
  (let ((result nil) (computed nil))
    (lambda () (unless computed
                 (setq result (funcall proc)
                       computed t))
               result)))

(defmacro delay (expr)
  `(memo-proc (lambda () ,expr)))

(defun force (thunk)
  (funcall thunk))

(defmacro make-stream (head tail)
  `(cons (delay ,head) (delay ,tail)))
```

Streams in Common Lisp

```
(defun head (stream)
  (force (car stream)))

(defun tail (stream)
  (force (cdr stream)))

(defvar      +stream-done+ '#:stream-done)

(defconstant +the-empty-stream+ nil)

(defun empty-stream-p (stream)
  (or (eq stream +the-empty-stream+)
      (eq (head stream) +stream-done+)))
```

Streams in Common Lisp

```
(defun map-stream (proc stream)
  (cond ((empty-stream-p stream) +the-empty-stream+)
        (t (make-stream (funcall proc (head stream))
                        (map-stream proc
                                    (tail stream))))))

(defun for-each (proc stream)
  (loop
      for str = stream then (tail str)
      until (empty-stream-p str)
      do (funcall proc (head str)))
  (funcall proc +stream-done+))
```

Fixing for-each

```
(defun make-stream-ref (stream)
  (list stream))
(defmacro use-once (place)
  `(prog1 ,place (setf ,place nil)))
(defmacro get-stream (stream-ref)
  `(use-once (car ,stream-ref)))

(defun %for-each (proc ref)
  (loop
      for stream = (get-stream ref) then (tail stream)
      until (empty-stream-p stream)
      do (funcall proc (head stream)))
  (funcall proc +stream-done+))

(defmacro for-each (proc stream)
  `(%for-each ,proc (make-stream-ref ,stream)))
```

Streams from Queues

```
(defun make-stream-from-queue (queue)
  "Make a stream by popping from a queue"
  (make-stream (dequeue queue :wait t)
               (make-stream-from-queue queue)))

(defun make-stream-from-queues (queues)
  "Make a stream popping same item from all queues."
  (make-stream
   (loop with stories = '()
       for queue in queues
       do (pushnew (dequeue queue :wait t) stories)
       finally
         (if (> (length stories) 1)
             (error "Queues not in sync: ~a" stories)
           (return (first stories))))
   (make-stream-from-queues queues)))
```

Classifying Streams

```
(defmethod classify-stream ((classifier classifier)
                            stream)
  (map-stream #'(lambda (story)
                  (classify classifier story)
                  story)
              stream))

(defun classify-stream-greedily
           (classifier stream-ref result-queue)
  "Fetch stories from stream greedily and place them
   on the result queue.  Fetching will cause the
   classification of each story."
  (for-each #'(lambda (story)
                (enqueue result-queue story))
    (classify-stream classifier
                     (get-stream stream-ref))))
```

Classifying Streams

```
(defmethod run-classify-stream
           ((manager classifier-manager))
  "A lot like classify-stream-greedily. Fetch stories
   from the inbox as fast as we can and place them in
   the outbox of the destination."
  (with-slots (source classifier destination) manager
    (let* ((in-queue  (outbox source))
           (in-stream (make-stream-from-queue
                       in-queue))
           (out-queue (inbox destination)))
      (for-each #'(lambda (story)
                    (enqueue out-queue story))
        (classify-stream classifier
                    (make-stream-ref in-stream))))))
```

Serial Streams Classifier

```
(defmethod classify-stream
           ((classifier serial-classifier) stream)
  (loop
      for pstream = stream then pipeline
      for child in (children classifier)
      for result-queue = (new-queue)
      for pipeline = (make-stream-from-queue
                      result-queue)
      do
        (run-in-thread 'classify-stream-greedily
            :name   "Serial child"
            :params (list child
                          (make-stream-ref pstream)
                          result-queue))
      finally
        (return pipeline)))
```

Parallel Streams Classifier

```
(defmethod classify-stream
           ((classifier parallel-classifier) stream)
  (loop
      for child in (children classifier)
      for result-queue = (new-queue)
      do
       (run-in-thread 'classify-stream-greedily
           :name   "Parallel child"
           :params (list child
                         (make-stream-ref stream)
                         result-queue))
      collect result-queue into queues
      finally
        (return (make-stream-from-queues queues)))))
```

Multi Streams Classifier

```
(defclass multi-classifier (classifier)
  ((name     :initform “Multi Classifier”)
   (children :initform nil :initarg :children
    :documentation “Multiple instances of same classifier”))
  (:documentation “Run multiple instances of a classifier
    concurrently.  All read from same input stream
    and output is guaranteed to maintain ordering.”))


(defun split-stream-to-queues (steam queues)
  (let ((index 0) (count (length queues)))
    (for-each (lambda (story &aux (queue (elt queues index)))
                (setq index (mod (1+ index) count))
                (enqueue queue story)) stream)))

(defun merge-stream-from-queues (queues &optional (idx 0)
                                 (cnt (length queues))
                                 (new-idx (mod (1+ idx) cnt)))
  (make-stream (dequeue (elt idx queues) :wait t)
               (merge-stream-from-queues queues new-idx)))
```

Multi Streams Classifier

```
(defmethod classify-stream
           ((classifier multi-classifier) stream)
  (loop
      for child in (children classifier)
      for work-q = (new-queue)
      for work-s = (make-stream-from-queue work-q)
      for result-q = (new-queue)
      do
        (run-in-thread ‘classify-stream-greedily
            :name “Multi Child”
            :params (list child
                          (make-stream-ref work-s)
                          result-q))
      collect work-q   into work-queues
      collect result-q into result-queues
      finally
        (run-in-thread ‘split-stream-to-queues
            :params (list stream work-queues))
        (merge-stream-from-queues result-queues)))
```

Streams Framework

- Pro: Simple interface to classifier
- Pro: Full control over dependencies
- Pro: Full multi-threading, pipelining
- Pro: Multiple stories classified at once
- Pro: Extensions like multi-classifier

We've met the challenge

Notes / Improvements

- memo-proc isn't thread safe
- memoization actually required here
- buffer-bloat / length-limited queues
- distributed classification

Reference / Contact

- SICP Chapter 3, Section 4
- http://groups.csail.mit.edu/mac/classes/6.001/abelson-sussman-lectures/

  Lectures 6A and 6B
- Franz Allegro Common Lisp 9.0
- Jason Cornez: jcornez@ravenpack.com