MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science

6.5150 Spring 2024
Problem Set 9

Issued: Wed. 16 April 2024   **Due: Fri. 3 May 2024**

Note: This is the last problem set for this term. You have a term
**   project to work on. Also, this problem set is optional for**
**   students enrolled in 6.5151.**

Readings:

  Online MIT/GNU Scheme Documentation,
     Section 2.3: Dynamic Binding – parameterize and fluid–let
     Section 12.4: Continuations   – call–with–current–continuation

  Here is a nice paper about continuations and threads:
      http://repository.readscheme.org/ftp/papers/sw2003/Threads.pdf

  In fact, there is an entire bibliography of stuff about this on:
      http://library.readscheme.org/page6.html

Code: load.scm ghelper.scm utils.scm time-share.scm schedule.scm
      syntax.scm rtdata.scm interp-actor.scm repl.scm

The Actor Model of Concurrency

We have examined traditional time–sharing in lecture, and the
propagator system we have played with is one way to think about
multiple computing agents acting in parallel: each propagator can be
thought of as an independent computing agent. But we didn't actually
implement the propagator system so that it could work in parallel. To
make that work we would have to arrange that cells could serialize
update requests so that they would not suffer from reader–writer
problems. Also, the propagator system, as implemented, manipulates
worldviews by global side–effects on the status of premises. In a
concurrent system it would be appropriate for the worldview to be
represented separately for each query process. Rebuilding the
propagator system to work well concurrently is an excellent project,
but here we go down another path for concurrency.

In 1973 Carl Hewitt came up with an idea for formulating concurrency,
based on message passing, called "actors." (Yes, that is the same
Carl Hewitt who had the idea of match combinators.) Very crudely,
Hewitt's idea is that when you call an actor (send it a message) the

call immediately returns to the caller, but the message is put on a
queue of work for the actor to do. Each actor processes the elements
of its input queue, perhaps changing its local state as it does so.
In Hewitt's actors the state update is accomplished by the actor
"replacing itself" with a new behavior. If the intention of the
caller was to receive a reply it would have to pass its return
continuation as an argument to the called actor as part of its calling
message.

In Hewitt's ideal system there were no computational objects that were
not actors, including integers. So to add "2" to "3" you send an
"add" message with a return address to "2". You then receive a
2-adder to which you send a "3" and a return address which will
receive "5". This made a very elegant object-oriented system, where
everything was implemented by message passing. See the article
http://en.wikipedia.org/wiki/Actor_model for more information.

Here I will be less orthodox and consider an actor-inspired system
that adds an actor-like procedure to an ordinary Scheme-like lambda-
calculus interpreter. This is more to my (GJS) liking, because it
does not require everything to be an actor -- I hate grand theories
and magic bullets that restrict a programmer to do things by someone's
grand theory. The languages Erlang and Scala both provide mechanisms
to support actor-like behaviors.

So, let's get into it. We provide an embedded actor interpreter
system -- you can load it with (load "load"). We initialize
this interpreter with (init) and if we get an error, we can reenter it
with (go). It provides the prompt "eval>".

This actor simulator is a work in progress. THERE ARE BUGS! However
there is much that we can learn from this system.

In this system we can run ordinary programs in the usual way:

```
     eval> (define fib
             (lambda (n)
               (if (< n 2)
                   n
                   (+ (fib (- n 1))
                      (fib (- n 2)))))))

     eval> (fib 10)
     55
```

But we can also define the Fibonacci computer in terms of actors,
using "alpha expressions" rather than "lambda expressions":

```
eval> (define fib
          (alpha (n c)
             (if (< n 2)
                 (c n)
                 (let ((x 'not-ready) (y 'not-ready))
                   (define wait-for-xy
                     (alpha (k)
                            (if (boolean/or
                                 (eq? x 'not-ready)
                                 (eq? y 'not-ready))
                                (wait-for-xy k)
                                (k #t))))
                   (fib (- n 1)
                        (lambda (v) (set! x v)))
                   (fib (- n 2)
                        (lambda (v) (set! y v)))
                   (wait-for-xy
                    (lambda (ignore) (c (+ x y)))))))))

eval> (fib 10 write-line)
55
done
```

Here fib is an actor that takes a number and a continuation to call
with the value. If the number is not less than 2 it sets up two state
variables, x and y, to capture the values of the recursive calls. It
also defines an actor procedure that busy-waits until x and y are
available. The continuations of the recursive calls are just
assignments to the capture variables. In cases where we don't want
to start a concurrent process, we use lambda rather than alpha.

Implementation

We will start with a simple interpreter to build an experimental actor
system. The changes to the interpreter system are quite minimal.
First, we add a special-form syntax for alpha expressions:

```
(define (alpha? exp) (tagged-list? exp 'alpha))

(define (alpha-parameters alpha-exp) (cadr alpha-exp))

(define (alpha-body alpha-exp)
  (let ((full-body (cddr alpha-exp)))
    (sequence->begin full-body)))
```

We did not change or add to the syntax for definitions, so expressions of the form

```
(define (foo x y) <body>)
```

define ordinary compound procedures with lambda. Perhaps we should have a defactor?

Since eval is defined as generic we need to produce a dispatch for alpha expressions that makes an actor procedure, analogous to a lambda-specified procedure:

```
(defhandler eval
  (lambda (expression environment)
    (make-actor-procedure
     (alpha-parameters expression)
     (alpha-body expression)
     environment))
  alpha?)
```

Now, we need to specify the run-time data structures for such actor procedures:

```
(define-record-type actor-procedure
    (make-actor-procedure% vars bproc env task-queue runnable)
     actor-procedure?
  (vars   actor-parameters)
  (bproc actor-body)
  (env    actor-environment)
  (task-queue get-actor-task-queue set-actor-task-queue!)
  (runnable actor-runnable? set-actor-runnable!))

(define (make-actor-procedure vars bproc env)
  (make-actor-procedure% vars bproc env (queue:make) #f))
```

Note that actor procedures differ from ordinary compound procedures produced by lambda expressions, in that actor procedures have a task queue and a bit that is true only if the actor is scheduled to run.

Next, we augment the apply generic operation to handle actor
procedures:

```
(defhandler apply
  (lambda (actor operands calling-environment)
    (if (not (= (length (actor-parameters actor))
                (length operands)))
        (error "Wrong number of operands supplied"))
    (let ((arguments
            (map (lambda (parameter operand)
                   (evaluate-procedure-operand
                     parameter operand calling-environment))
                 (actor-parameters actor)
                 operands)))
      (add-to-tasks! actor
                     (lambda ()
                       (eval (actor-body actor)
                             (extend-environment
                              (map procedure-parameter-name
                                   (actor-parameters actor))
                              arguments
                              (actor-environment actor))))))
    'actor-applied)
  actor-procedure?)
```

This is just like an ordinary compound procedure except that we defer
the actual work to be done by the actor by adding it to the tasks.
Note that we made a choice here to defer only the execution of the
body. Is this choice a good one? Suppose we wanted to also defer the
evaluation of the operands? We'll come back to these questions later.

Finally, we have to provide a scheduler. In this simple scheduler
there is a task queue for each individual actor and there is a
runnable queue for actors that have non-empty task queues. A task is
represented by a thunk.

```
(define runnable-actors)

(define (init-actors)
  (set! runnable-actors (queue:make)))
```

The procedure switch-threads is the scheduler. It either returns a
task to be performed or it signals that there are no tasks to be
performed or that a runnable actor actually has no work to do.

If work is preempted by time sharing, the continuation of that work is
also put on the runnable-actors queue, and turned into a thunk, so it
can be executed as if it were a task.

```scheme
(define (switch-threads working-continuation)
  (define (get-task)
    (if (queue:empty? runnable-actors)
        'nothing-to-do
        (let ((actor (queue:get-first! runnable-actors)))
          (if (actor-procedure? actor)
              (let ((actor-task-queue (get-actor-task-queue actor)))
                ;; If there are no more tasks for this actor,
                ;; it is not runnable.
                (if (queue:empty? actor-task-queue)
                    (begin (set-actor-runnable! actor #f)
                           'try-again)       ; Nothing for actor.
                    (let ((task (queue:get-first! actor-task-queue)))
                      ;; Got the first task, and took it off the queue.
                      ;; If there are no more tasks for this actor,
                      ;; it is not runnable.
                      ;; If there are more tasks for this actor,
                      ;; requeue it.
                      (if (queue:empty? actor-task-queue)
                          (set-actor-runnable! actor #f)
                          (queue:add-to-end! runnable-actors actor))
                      ;; Return task to be executed.
                      task)))
              (lambda () (actor 'go))))))
  (let lp ((task (atomically get-task)))
    (cond ((eq? task 'nothing-to-do)
           (working-continuation 'resume))
          ((eq? task 'try-again)
           (lp (atomically get-task)))
          (else
           (task)
           (working-continuation 'resume)))))
```

To add a task to an actor we must add it to the actor's task
queue and we must make the actor runnable. This must all be
done atomically.

```scheme
(define (add-to-tasks! actor task)
  (atomically
    (lambda ()
      (queue:add-to-end! (get-actor-task-queue actor) task)
      (add-to-runnable! actor)))
  'task-added)

(define (add-to-runnable! actor)
  (if (actor-runnable? actor)
      'already-runnable
      (begin (set-actor-runnable! actor #t)
             (queue:add-to-end! runnable-actors actor)
             'made-runnable)))
```

An actor that is already on the runnable-actors queue has its runnable
bit set. This allows the scheduler to avoid putting multiple
references to the same actor on the runnable actors queue.

The read-eval-print loop has to be updated to correctly start the
system:

```scheme
  (define (init)
    (set! the-global-environment
          (extend-environment '() '() the-empty-environment))
    (init-actors)                                ;in schedule.scm
    (repl))

  (define (repl)
    (if (eq? the-global-environment 'not-initialized)
        (error "Interpreter not initialized. Run (init) first."))
    (setup-time-sharing)
    (let lp ((input (read)))
      (let ((output (eval input the-global-environment)))
        (write-line
         `(";pending:" ,(length (queue:front-ptr runnable-actors))))
        (display ";==>: ")
        (write-line output))
      (lp (read))))

  (define (go) (repl))
```

There is a preemptive timer interrupt that runs this whole mess.
There is an MIT/GNU Scheme specific detail. register-timer-event is
the MIT/GNU Scheme mechanism for delivering a timer interrupt. When
the time specified by its first argument expires, it invokes the
second argument.
This part of the system was very hard to get right: GJS was up all
night debugging it!

```scheme
    (define root-continuation)

    (define (setup-time-sharing)
      (call-with-current-continuation
       (lambda (k)
         (set! root-continuation k)
         (if time-sharing-enabled?
             (start-time-sharing)))))

    (define time-sharing:quantum 10)
    (define time-sharing-enabled? #t)
    (define time-sharing? #f)
```

Time sharing among the actors is accomplished as follows:

```scheme
    (define (start-time-sharing)
      (set! time-sharing? time-sharing-enabled?)
      (define (setup-interrupt)
        (if time-sharing?
            (register-timer-event time-sharing:quantum
                                  on-interrupt)))
      (define (on-interrupt)
        (setup-interrupt)
        (call-with-current-continuation switch-threads))
      (setup-interrupt))                ; Initialize interrupt

    (define (stop-time-sharing)
      (set! time-sharing? #f))
```

This code calls the scheduler, switch-threads on each timer interrupt.

There are a few more things that are necessary to make this system.
All user variables that are modified must be protected from
reader-writer problems.

```
(define (set-variable-value! var val env)
  (let plp ((env env))
    (if (eq? env the-empty-environment)
        (error "Unbound variable -- SET!" var)
        (let scan
            ((vars (vector-ref env 0))
             (vals (vector-ref env 1)))
          (cond ((null? vars) (plp (vector-ref env 2)))
                ((eq? var (car vars))
                 (double-check-lock
                  (lambda () (eq? var (car vars)))
                  (lambda ()
                    (set-car! vals val))
                  (lambda () (plp env))))
                (else (scan (cdr vars) (cdr vals)))))))))
```

The assignment of the variable is guaranteed to be atomic. There is a
similar, but somewhat hairier problem with define-variable!.

Problem 9.1:

Here is a simple program that is implemented with actors.

```
(define (foo n)
  (define (buzz m)
    (if (not (= m 0)) (buzz (- m 1))))
  (define iter
    (alpha (l i)
       (if (not (= i 0))
           (begin
             (if (eq? l 'a) (buzz (* 100 i)) (buzz (* 100 (- n i))))
             (pp (list l i))
             (iter l (- i 1))))))
  (iter 'a n)
  (iter 'b n))
```

When I ran this program I got the following output:

```
eval> (foo 10)
actor-applied

eval> (b 10)
(b 9)
(b 8)
(a 10)
(b 7)
(b 6)
(a 9)
(b 5)
(b 4)
(a 8)
(b 3)
(a 7)
(a 6)
(b 2)
(a 5)
(b 1)
(a 4)
(a 3)
(a 2)
(a 1)
```

You may get a different order than I got. Why do we get this output
in such a strange order? Explain what you see here.

Problem 9.2:

Note that we have made a choice in the handler for application of
actor procedures: we defer only the execution of the body. Is this
choice a good one? Suppose we wanted to also defer the evaluation of
the operands?

a. Explain, in a short clear paragraph your opinion on this matter.

b. Write an apply handler for actors that defers the evaluation of
   operands. Demonstrate it. Does it have the properties you expect?

c. Would it be advantageous to supply another kind of object, perhaps
   defined by a "beta expression" that has this behavior?

Problem 9.3:

The procedure double-check-lock is used in set-variable-value! and
define-variable!.

```
(define (double-check-lock check do if-not)
  (let ((outside
          (atomically
            (lambda ()
              (if (check)
                  (begin (do)
                         (lambda () 'ok))
                  if-not)))))
    (outside)))
```

Why is it needed? Explain how it works.

Problem 9.4:

The first "actor implementation" of the Fibonacci procedure was

```
(define fib1
  (alpha (n c)
    (if (< n 2)
        (c n)
        (let ((x 'not-ready) (y 'not-ready))
          (define wait-for-xy
            (alpha (k)
                   (if (boolean/or (eq? x 'not-ready)
                                   (eq? y 'not-ready))
                       (wait-for-xy k)
                       (k #t))))
          (fib1 (- n 1) (lambda (v) (set! x v)))
          (fib1 (- n 2) (lambda (v) (set! y v)))
          (wait-for-xy (lambda (ignore) (c (+ x y)))))))))
```

a. Why is this so slow compared with the traditional version of fib
defined as a doubly-recursive function in the same interpreter?

Perhaps it is worthwhile to consider the following alternative
implementation of Fibonnaci:

```
(define fib2
  (alpha (n c)
    (if (< n 2)
        (c n)
        (let ((x 'not-ready) (y 'not-ready))
          (define check-if-done
            (lambda ()
              (if (boolean/or (eq? x 'not-ready)
                              (eq? y 'not-ready))
                  #f
                  (c (+ x y)))))
          (fib2 (- n 1)
                (lambda (v)
                  (set! x v)
                  (check-if-done)))
          (fib2 (- n 2)
                (lambda (v)
                  (set! y v)
                  (check-if-done)))))))
```

The essential difference is that the end test in the first version is
implemented as an actor loop, whereas in the second version it is
implemented as a Scheme procedure.

b. The fib2 implementation happens to have a subtle bug. Sometimes it
prints the result multiple times: try (fib2 10 write-line) a few
times, if you don't see ths problem (it is a heisenbug -- see
Wikipedia) try other numbers, like 8,9,11. Can you explain why this
occurs? How could you fix it? Caution: this is very hard.

Futures

The procedures that were illustrated in Problem 9.4 share an idea. An
actor is started; when it finishes it sets a return value in the
caller. The caller must check when the results of pending
computations are completed for it to proceed to use those results.
This pattern should be abstracted. One way is by the introduction of
"future"s. (This abstraction was introduced by Robert H. Halstead,
Jr. in the early 1980's. Sometimes futures are called "promises", but
this word usually refers to delay thunks.) With futures, the code for
Fibonacci looks like:

```
(define fib
  (alpha (n c)
    (if (< n 2)
        (c n)
        (let ((xp (future (lambda (k) (fib (- n 1) k))))
              (yp (future (lambda (k) (fib (- n 2) k)))))
          (wait xp
            (lambda (x)
              (wait yp
                (lambda (y)
                  (c (+ x y)))))))))))

eval> (fib 10 write-line)
actor-applied
55
```

Notice that the future converts the continuation into an object whose
fulfillment one can wait for.

A future is a mutable data structure. It contains a continuation, a
"done" flag, a value. When a future gets fulfilled, its value is
stored, and the done flag is set.

Problem 9.5:

Here we implement futures. The implementation should be added to the
guest interpreter.

a. Your first job is to construct the data structure for a future. We
   recommend using record structures for this.

b. Implement procedures FUTURE and WAIT. Make sure that your WAIT
   does not hang: it should return immediately but put a continuation
   on a queue of things to do. Assume here that a future is waited on
   only once. Make sure that you protect critical regions with
   ATOMICALLY.

c. What goes wrong with your implementation if the assumption is violated?

d. Improve your implementation to relax the assumption.