Matthew Johnson
February 21, 2013

6.945: Symbolic Programming

Problem Set 1

Problem 1.1:

See the code in Listing 1 and the trace in Listing 2.

```
(define-syntax remove-advice
  (syntax-rules ()
        ((remove-advice p)
         (if (eq-get p 'old-version)
          (begin
            (set! p (eq-get p 'old-version))
            'done)
          (warn "procedure was not advised"))
         )))

(define-syntax advise-unary
  (syntax-rules ()
        ((advise-unary p wrapper)
          (define p
            (let ((saved-p p) (ewrapper wrapper))
              (let ((new-p (named-lambda (p x) (ewrapper 'p saved-p x))))
                (if (procedure-of-arity? p 1)
                  (begin
                    (eq-put! new-p 'old-version saved-p)
                    new-p)
                  (begin
                    (warn "procedure not unary")
                    saved-p))))))))

(define-syntax advise-binary
  (syntax-rules ()
        ((advise-binary p wrapper)
          (define p
            (let ((saved-p p) (ewrapper wrapper))
              (let ((new-p (named-lambda (p x y) (ewrapper 'p saved-p x y))))
                (if (procedure-of-arity? p 2)
                  (begin
                    (eq-put! new-p 'old-version saved-p)
                    new-p)
                  (begin
                    (warn "procedure not binary")
                    saved-p))))))))

(define-syntax advise-nary
  (syntax-rules ()
        ((advise-nary p wrapper)
          (define p
            (let ((saved-p p) (ewrapper wrapper))
              (let ((new-p (named-lambda (p . arglist) (ewrapper 'p saved-p arglist))))
                (eq-put! new-p 'old-version saved-p)
                new-p))))))
```

Listing 1: Code for Problem 1.1.

```
1 ]=> (advise-binary cons (lambda (name oldf x y)
                      (if (< (random 1.0) 0.5)
                        (oldf x y)
;Value: cons
1 ]=> (cons 1 2)
;Value 10: (2 . 1)
1 ]=> (cons 1 2)
;Value 11: (2 . 1)
1 ]=> (cons 1 2)
;Value 12: (2 . 1)
1 ]=> (cons 1 2)
;Value 13: (1 . 2)
1 ]=> (cons 1 2)
;Value 14: (1 . 2)
1 ]=> (remove-advice cons)
;Value: done

1 ]=> (advise-nary + (lambda (name oldf args)
                 (if (any inexact? args)
                   (warn "inexact addition"))
                 (apply oldf args)))
;Value: +
1 ]=> (+ 1 2)
;Value: 3
1 ]=> (+ 1.0 2)
;Warning: inexact addition
;Value: 3.
1 ]=> (remove-advice +)
;Value: done

1 ]=> (advise-unary cons (lambda (name oldf x) '()))
;Warning: procedure not unary
;Value: cons
1 ]=> (remove-advice +)
;Warning: procedure was not advised
;Unspecified return value
1 ]=> (+ 1 2)
;Value: 3
```

Listing 2: Trace for Problem 1.1.

Problem 1.2:

a. Yes, I think it is necessary to break tail recursion because printing out the “Leaving” message and returning value means that the recursive call is not the last thing the wrapped function does. The data that must be added to the stack include the return addresses into the wrapper procedure body, and the number of such return addresses added to the stack corresponds to the number of times the “Leaving” statement must be printed before returning to the original caller.

b. See the code in Listing 3 and the trace in Listing 4. Since my solution dynamically un-advises the procedure, if the procedure calls another procedure which in turn calls the first procedure, this tracing mechanism won’t reveal that second “distinct” call into this procedure. However, it seems necessary to remove the advice wrapper to allow tail recursion.

```
(define (simpler-full-trace-wrapper procname proc args)
  (newline)
  (display ";Entering ") (write procname)
  (display " with ") (write args)
  (let ((val (fluid-let ((procname proc)) (apply proc args))))
    (newline)
    (display ";Leaving ") (write procname)
    (display " Value=") (write val)
    val))
```

Listing 3: Code for Problem 1.2.

```
1 ]=> (advise-nary fact simpler-full-trace-wrapper)
;Value: fact
1 ]=> (fact 5)
;Entering fact with (5)
;Leaving fact Value=120
;Value: 120
```

Listing 4: Trace for Problem 1.2.

Problem 1.3:

a. It's really not clear to me what kinds of cryptographic mechanisms would be reasonable here, so here are some assumptions that might yield a reasonable model:

   – There must be some restricted access to "internal" data structures (the tables accessed by the authorization wrapper with eq-put and eq-get, though the code itself must also be un-editable). If not, a user could write into both types of table so that any security procedure that relies only on checking the data in those tables would be compromised. Therefore I'll assume that the "procedure authorization-keys" tables and the "user authorizations" tables are both *read-only* to the user. (Other combinations of readability and writeability that make sense don't seem to be any different.)
   – It's hard to invert the hash function.
   – Each user has a secret key (or password). (Alternatively, the usernames could be secret, but that's weird.)
   – Users can't use remove-advice.

   In the original system, a user without access to the `sin` function could gain access simply by by running `(define user-id gjs)`. With these assumptions, we can add a simple mechanism to prevent users from such circumventions of the authorization system even if they can read the authorization system's data structures: all we need to do is change the user id check to one based on a hard-to-invert function of the secret key.

   User IDs seemed redundant, so now permissions are associated with secret keys.

   See the code in Listing 5 and the trace in Listing 6.

   This modification is really simplistic and contrived, but I can't think of something that makes more sense.

b. In addition to restricting eq-put action on the data structures relevant to the authorization system and restricting the ability to remove the authorization advice, subprocess also shouldn't be allowed unless users are meant to have general shell access and TCP sockets should be restricted so that malicious users can't intercept other users' connections (though that may be prevented by the OS's security for port binding). In general it seems that OS-level or even hardware-level (hypervisor) sandboxing provide good authorization mechanisms to prevent unauthorized data access, so the restrictions that we'd want inside a Scheme environment depend on what utilization model we want to enforce.

```scheme
(define (authorization-wrapper procname proc args)
  (cond ((memq (eq-get proc 'authorization-key)
               (or (eq-get (string-hash (md5-string secret-key)) 'authorizations)
                   '()))
         (apply proc args))
        (else
          (error "Unauthorized access" procname))))

;; here's how a superuser might restrict access to a function and add
;; permissions for a user (usually eq-put! can't be called on these
;; structures!)

(eq-put! sin 'authorization-key 'ok-to-sin)
(advise-nary sin authorization-wrapper)
(eq-put! (string-hash (md5-string "louisreasonerkey")) 'authorizations
         '(ok-to-sin))
```

Listing 5: Code for Problem 1.3.

```scheme
;; here's how a user might use his powers of trigonometry
1 ]=> (define secret-key "louisreasonerkey")
;Value: secret-key
1 ]=> (sin 0)
;Value: 0

;; here's how a user might fail at the same
1 ]=> (define secret-key "bjarne")
;Value: secret-key
1 ]=> (sin 0)
;Unauthorized access sin
```

Listing 6: Trace for Problem 1.3.

Problem 1.4:

See the code in Listing 7 and the trace in Listing 8. There is almost certainly also a good way to hash argument lists to accomplish the same tasks; I think there might be pitfalls there, though.

```scheme
(define (all-list-elements-eqv l1 l2)
  (or (and (null? l1) (null? l2))
      (and (not (or (null? l1) (null? l2)))
           (eqv? (car l1) (car l2))
           (all-list-elements-eqv (cdr l1) (cdr l2)))))

(define assvl (association-procedure all-list-elements-eqv car))

(define (memo-wrapper-nary procname proc args)
  (let ((seen (assvl args (eq-get proc 'old-values))))
    (if seen
      (cdr seen) ;; for testing, replace with (begin (display ";got cached") (cdr seen))
      (let ((v (apply proc args)))
        (eq-put! proc
          'old-values
          (cons (cons args v)
                      (eq-get proc 'old-values)))
        v))))
```

Listing 7: Code for Problem 1.4.

```
1 ]=> (eq-put! + 'old-values '())
;Value 2: #[arity-dispatched-procedure 2]
1 ]=> (advise-nary + memo-wrapper-nary)
;Value: +
1 ]=> (+ 1 2 3)
;Value: 6
1 ]=> (+ 1 2 3)
;got cached
;Value: 6
```

Listing 8: Trace for Problem 1.4.