Matthew Johnson
February 27, 2013

6.945: Symbolic Programming

Problem Set 2

Problem 2.1:

See the code below and the trace in Listing 1.

```scheme
;;; (sequence:equal? <sequence-1> <sequence-2>)

;; This implementation uses the other generic functions and it is nice, but
;; might go against the spirit of the assignment
;; (see below for another implementation)
(define (all-eq? . args)
  (or (null? args)
      (let ((v (car args)))
        (for-all? (cdr args) (lambda (x) (eq? x v))))))

(define (sequence:equal? . seqs)
  (if (< (length seqs) 2)
    (error "Need at least two sequences for equal?" seqs)
    (and
      (for-all? (cdr seqs) (sequence:type (car seqs)))
      (sequence:fold-right boolean/and #t (apply sequence:map (cons all-eq? seqs))))))

;; This implementation may be more in line with the original intent of the
;; assignment
(define sequence:equal? (make-generic-operator 2))

(define (vector=? v1 v2)
  (and (= (vector-length v1) (vector-length v2))
       (let ((len (vector-length v1)))
        (let loop ((i 0))
          (cond ((= i len) #t)
                ((not (eq? (vector-ref v1 i) (vector-ref v2 i))) #f)
                (else (loop (+ i 1))))))))

(define (list=? l1 l2)
  (every eq? l1 l2))

(define (compose-both-args f g1 g2)
  (lambda (x y) (f (g1 x) (g2 y))))

(defhandler sequence:equal? string=?  string? string?)
(defhandler sequence:equal? list=?    list?   list?)
(defhandler sequence:equal? vector=?  vector? vector?)

;;; (sequence:generate <sequence-type> <n> <function>)
(define ((drop-first-arg f) . args)
  (apply f (cdr args)))

(define (make-initialized-string n func)
  (let ((s (make-string n)))
    (let loop ((i 0))
      (if (= i n)
        s
        (begin
          (string-set! s i (func i))
          (loop (+ i 1)))))))
```

```scheme

(defhandler sequence:generate (drop-first-arg make-initialized-string)
            (is-exactly string?) exact-integer? procedure?)
(defhandler sequence:generate (drop-first-arg make-initialized-list)
            (is-exactly list?)   exact-integer? procedure?)
(defhandler sequence:generate (drop-first-arg make-initialized-vector)
            (is-exactly vector?) exact-integer? procedure?)

;;; (sequence:construct <sequence-type> <item-1> ... <item-n>)
(define (sequence:construct type? . items)
  (sequence:generate type? (length items) (lambda (i) (list-ref items i))))

;;; (sequence:map <function> <seq-1> ... <seq-n>)
;; Using sequence:ref on list-backed sequences will waste some dereferences,
;; but the benefit is simpler code.
;; Note we can mix sequences here by simply removing the check that all types
;; are the same.
(define (sequence:map func . seqs)
  (if (null? seqs)
    (error "Need at least one sequence for map"))
  (let ((type? (sequence:type (car seqs)))
        (size  (sequence:size (car seqs))))
    (if (not (for-all? (cdr seqs) type?))
      (error "All sequences for map must be of the same type" seqs))
    (if (not (for-all? (cdr seqs) (lambda (x) (= (sequence:size x) size))))
      (error "All sequences for map must be of the same size" seqs))

    (define (index-func i)
      (apply func (map (lambda (s) (sequence:ref s i)) seqs)))
    (sequence:generate type? size index-func)))

;;; (sequence:for-each <procedure> <seq-1> ... <seq-n>)
;; This constructs an extra sequence, but it's so much easier to implement in
;; terms of map
(define (sequence:for-each . args)
  (apply sequence:map args)
  #!unspecific)

;;; (sequence:filter <sequence> <predicate>)
(define sequence:filter (make-generic-operator 2))

(define ((switch-args f) x y) (f y x))

;; since we have to pull out elements and then construct the contiguous types
;; at the end, coercion to a list intermediate is appropriate here
(define (vector-filter predicate v)
  (list->vector (filter predicate (vector->list v))))

(define (string-filter predicate s)
  (list->string (filter predicate (string->list s))))

(defhandler sequence:filter (switch-args string-filter)   string? procedure?)
(defhandler sequence:filter (switch-args filter)          list?   procedure?)
(defhandler sequence:filter (switch-args vector-filter)   vector? procedure?)

;;; (sequence:get-index <sequence> <predicate>)
(define sequence:get-index (make-generic-operator 2))

(define (list-get-index l predicate)
```

```
  (define (loop l i)
    (cond ((null? l) #f)
          ((predicate (car l)) i)
          (else (loop (cdr l) (+ i 1)))))
  (loop l 0))

(define (contiguous-sequence-get-index s predicate)
  (let ((size (sequence:size s)))
    (define (loop i)
      (cond ((= i (- size 1)) #f)
            ((predicate (sequence:ref s i)) i)
            (else (loop (+ i 1)))))
    (loop 0)))

(defhandler sequence:get-index contiguous-sequence-get-index    string? procedure?)
(defhandler sequence:get-index list-get-index                   list?   procedure?)
(defhandler sequence:get-index contiguous-sequence-get-index    vector? procedure?)

;;; (sequence:get-element <sequence> <predicate>)
(define (sequence:get-element sequence predicate)
  (sequence:ref sequence (sequence:get-index sequence predicate)))

;;; (sequence:fold-right <function> <initial> <sequence>)
(define sequence:fold-right (make-generic-operator 3))

(define (contiguous-sequence-fold-right op init seq)
  (let loop ((result init)
             (i (- (sequence:size seq) 1)))
    (if (< i 0)
      result
      (loop (op (sequence:ref seq i) result)
            (- i 1)))))

(defhandler sequence:fold-right contiguous-sequence-fold-right  procedure? any? string?)
(defhandler sequence:fold-right fold-right                      procedure? any? list?)
(defhandler sequence:fold-right contiguous-sequence-fold-right  procedure? any? vector?)

;;; (sequence:fold-left <function> <initial> <sequence>)
(define sequence:fold-left (make-generic-operator 3))

(define (contiguous-sequence-fold-left op init seq)
  (let ((size (sequence:size seq)))
    (let loop ((result init)
               (i 0))
      (if (= i size)
        result
        (loop (op result (sequence:ref seq i))
              (+ i 1))))))

(defhandler sequence:fold-left contiguous-sequence-fold-left   procedure? any? string?)
(defhandler sequence:fold-left fold-left                       procedure? any? list?)
(defhandler sequence:fold-left contiguous-sequence-fold-left   procedure? any? vector?)
```

```
1 ]=> (sequence:equal? '(1 2 3) '(1 2 3))
;Value: #t
1 ]=> (sequence:equal? '(1 2 3) '(1 2 4))
;Value: #f

1 ]=> (sequence:generate string? 5 (lambda (i) #\a))
;Value 3: "aaaaa"
1 ]=> (sequence:generate vector? 5 (lambda (i) (* i 2)))
;Value 4: #(0 2 4 6 8)

1 ]=> (sequence:construct list? 1 2 3)
;Value 5: (1 2 3)

1 ]=> (sequence:map * '(1 2) '(3 4))
;Value 6: (3 8)
1 ]=> (sequence:map * #(1 2) #(3 4))
;Value 7: #(3 8)

1 ]=> (sequence:filter #(1 2 3 4 5) odd?)
;Value 10: #(1 3 5)

1 ]=> (sequence:get-element #(0 0 0 5 1 2 1) odd?)
;Value: 5
1 ]=> (sequence:get-index #(0 0 0 5 1 2 1) odd?)
;Value: 3

1 ]=> (sequence:fold-right cons '() #(1 2 3 4))
;Value 4: (1 2 3 4)
1 ]=> (sequence:fold-left cons '() #(1 2 3 4))
;Value 5: (((((() . 1) . 2) . 3) . 4)
```

Listing 1: Trace for Problem 2.1.

Problem 2.2:

It might be useful to have an equality comparison that can handle multiple types, though that can be accomplished simply by removing the type check which happens in the first clause of the conjunction in my first implementation (above). The same goes for my map and for-each implementations: they can handle multiple types simply by removing a check, since they're implemented in terms of other generics. I don't think others would be very useful.

See the revised specifications below and the code in Listing 2.

```
;;; (sequence:append <sequence-1> ... <sequence-n>)
;;; REVISED
;;;    Returns a new sequence of the same type as the first sequence argument,
;;;    formed by concatenating the elements of the given sequences.  The size
;;;    of the new sequence is the sum of the sizes of the given sequences.

;;; (sequence:equal? <sequence-1> <sequence-2>)
;;; REVISED
;;;    Returns #t if the elements of the sequence are the same (and in the same
;;;    order), otherwise returns #f

;;; (sequence:map <function> <seq-1> ... <seq-n>)
;;; REVISED
```

```
;;;    Requires that the sequences given are of the same size,
;;;    and that the arity of the function is n.  The ith element
;;;    of the new sequence is the value of the function applied to the
;;;    n ith elements of the given sequences. The type of the result is the
;;;    type of the first sequence.

;;; (sequence:for-each <procedure> <seq-1> ... <seq-n>)
;;; REVISED
;;;    Requires that the sequences given are of the same size,
;;;    and that the arity of the procedure is n.  Applies the
;;;    procedure to the n ith elements of the given sequences;
;;;    discards the value.  This is done for effect.
```

```
(define (string->vector s)
  (make-initialized-vector (string-length s) (lambda (i) (string-ref s i))))

(define (vector->string v)
  (make-initialized-string (vector-length v) (lambda (i) (vector-ref v i))))

(defhandler sequence:binary-append (compose-2nd-arg append string->list)          list?   string?)
(defhandler sequence:binary-append (compose-2nd-arg string-append list->string)   string? list?)
(defhandler sequence:binary-append (compose-2nd-arg vector-append string->vector) vector? string?)
(defhandler sequence:binary-append (compose-2nd-arg string-append vector->string) string? vector?)

(define (sequence:map func . seqs)
  (if (null? seqs)
    (error "Need at least one sequence for map"))
  (let ((type? (sequence:type (car seqs)))
        (size  (sequence:size (car seqs))))
    (if (not (for-all? (cdr seqs) (lambda (x) (= (sequence:size x) size))))
      (error "All sequences for map must be of the same size" seqs))

    (define (index-func i)
      (apply func (map (lambda (s) (sequence:ref s i)) seqs)))
    (sequence:generate type? size index-func)))

(define (sequence:equal? . seqs)
  (if (< (length seqs) 2)
    (error "Need at least two sequences for equal?" seqs)
    (sequence:fold-right boolean/and #t (apply sequence:map (cons all-eq? seqs)))))
```

Listing 2: Code for Problem 2.2

Problem 2.3:

There could be a big efficiency advantage: complex numbers are a good example, and potentially any computation based on manipulating mathematical group elements which could be backed by various representations (like SO(3) backed by quaternions or rotation matrices, or Gaussian distributions backed by covariance or information parameterizations) in which some operations are more efficient in some representations. With the positional argument signature we would more or less need to stick to this fixed-arity scheme for handlers to prevent ambiguity, and it makes the syntax and error checking a bit less clear. Also any generic procedure not defined using the ghelper.scm lookup table (e.g. sequence:append defined in terms of other generics) would need to handle the optional first argument stuff. Alternatively, we could rely on some kind of type inference system, but I don't know anything about those!

This optional argument acts pretty much like any other argument. We must extend defhandler to take procedures with arity n or n+1 given a procedure record specifying arity n, and the procedure returned by make-generic-operator needs logic so that, if the length of the argument list is n+1, we apply the procedure only to the real input arguments.

See the code below and the trace in Listing 3.

```
;;; IN ghelper.scm
(define (make-generic-operator arity #!optional name default-operation)
  (let ((record (make-operator-record arity)))
    (define (operator . arguments)
      (let ((input-arguments (if (= (length arguments) (+ arity 1)) ;; CHANGED
                               (cdr arguments)
                               arguments)))
        (if (not (= (length input-arguments) arity))
          (error "Wrong number of arguments for generic operator"
                 (if (default-object? name) operator name)
                 arity arguments))
        (apply (or (let per-arg
                     ((tree (operator-record-tree record))
                      (args arguments))
                     (let per-pred ((tree tree))
                       (and (pair? tree)
                            (if ((caar tree) (car args))
                              (if (pair? (cdr args))
                                (or (per-arg (cdar tree) (cdr args))
                                    (per-pred (cdr tree)))
                                (cdar tree))
                              (per-pred (cdr tree))))))
                   (if (default-object? default-operation)
                     (lambda args
                       (error "No applicable methods for generic operator"
                              (if (default-object? name) operator name)
                              args))
                     default-operation))
               input-arguments))) ;; CHANGED

    (hash-table/put! *generic-operator-table* operator record)
    operator))

(define (defhandler operator handler . argument-predicates)
  (let ((record
          (let ((record (hash-table/get *generic-operator-table* operator #f))
                (arity (length argument-predicates)))
            (if record
              (begin
                (if (not (or (= arity (operator-record-arity record)) ;; CHANGED
                             (= (- arity 1) (operator-record-arity record))))
                  (error "Incorrect operator arity:" operator))
                record)
              (error "Operator not known" operator)))))
    (set-operator-record-tree! record
                               (bind-in-tree argument-predicates
                                             handler
                                             (operator-record-tree record))))
  operator)
```

```
;;; IN generic-sequences.scm
(define sequence:coerce (make-generic-operator 2))
(define (id x) x)

; coerce from a list
(defhandler sequence:coerce (drop-first-arg list->vector)     (is-exactly vector?) list?)
(defhandler sequence:coerce (drop-first-arg list->string)     (is-exactly string?) list?)
(defhandler sequence:coerce (drop-first-arg id)               (is-exactly list?)   list?)
; coerce from a vector
(defhandler sequence:coerce (drop-first-arg vector->list)     (is-exactly list?)   vector?)
(defhandler sequence:coerce (drop-first-arg vector->string)   (is-exactly string?) vector?)
(defhandler sequence:coerce (drop-first-arg id)               (is-exactly vector?) vector?)
; coerce from a string
(defhandler sequence:coerce (drop-first-arg string->vector)   (is-exactly vector?) string?)
(defhandler sequence:coerce (drop-first-arg string->list)     (is-exactly list?)   string?)
(defhandler sequence:coerce (drop-first-arg id)               (is-exactly string?) string?)

(defhandler sequence:binary-append
            (lambda (x y) (sequence:binary-append (sequence:coerce vector? x) (sequence:coerce vector? y)))
            (is-exactly vector?) any? any?)
(defhandler sequence:binary-append
            (lambda (x y) (sequence:binary-append (sequence:coerce list? x) (sequence:coerce list? y)))
            (is-exactly list?) any? any?)
(defhandler sequence:binary-append
            (lambda (x y) (sequence:binary-append (sequence:coerce string? x) (sequence:coerce string? y)))
            (is-exactly string?) any? any?)
```

```
1 ]=> (sequence:binary-append list? (vector 'a 'b 'c) (list 'd 'e 'f))
;Value 3: (a b c d e f)
1 ]=> (sequence:binary-append vector? (vector 'a 'b 'c) (list 'd 'e 'f))
;Value 4: #(a b c d e f)
1 ]=> (sequence:binary-append (vector 'a 'b 'c) (list 'd 'e 'f))
;Value 5: #(a b c d e f)
```

Listing 3: Trace for Problem 2.3.

Problem 2.4:

It's a bad idea! It makes type-template matching really ambiguous: we may need to try matching all combinations of pushing types into the variadic list, and when there's an ambiguity we would need a convention to resolve it, like "the variadic list is greedy". It seems like things would get really messy for both the user and the maintainer, though.

Since we're using full predicates, we can always just pass list arguments and use list predicates insetad of variadic argument lists. Variadic argument lists may make more sense when types are tags and not full predicates.

If we really wanted to make this change (and we didn't want to settle for the list predicates thing), defhandler would need some new syntax for specifying that a predicate applies to the elements of the variadic list, and make-generic-operator would need to be able to check all the possible variadic interpretations against a variadic record specification.

Problem 2.5:

A. Here's a trace:

```
; I replaced the "same" case output with 'same
1 ]=> (list<? '(3 3) '(2 3))
;Value: same

1 ]=> (list<? '(2 2) '(4 1))
;Value: #t
1 ]=> (list<? '(4 1) '(2 2))
;Value: #t
```

Clearly our sort can be ambiguous and so we may end up with duplicates or, if we implement equality by something like

```
(and (not (list<? x y)) (not (list<? y x)))
```

we may end up throwing out distinct elements of the set.

B. Basically it's doing the same thing as the generic lookup table logic but implementing the logic in a procedure makes it harder to extend elsewhere (which is part of the point of this generics implementation!).

C. See the code below.

```
(define (constant val) (lambda args val))

(define (list<? list-1 list-2)
  (let ((len-1 (length list-1)) (len-2 (length list-2)))
    (cond ((< len-1 len-2) #t)
          ((> len-1 len-2) #f)
          ;; Invariant:  equal lengths
          (else
           (let prefix<? ((list-1 list-1) (list-2 list-2))
             (cond ((null? list-1) #f)  ; same
                   ((generic:less? (car list-1) (car list-2)) #t)
                   ((generic:less? (car list-2) (car list-1)) #f)
                   (else (prefix<? (cdr list-1) (cdr list-2))))))))))

(define generic:less? (make-generic-operator 2))

(let loop ((ordering (list null? boolean? char? number? symbol? string? vector? list?)))
  (if (> (length ordering) 1)
    (let ((small (car ordering))
          (big   (cadr ordering)))
      (defhandler generic:less? (constant #t) small big)
      (defhandler generic:less? (constant #f) big   small)
      (loop (cdr ordering)))))

(defhandler generic:less? char<?        char?   char?)
(defhandler generic:less? <             number? number?)
(defhandler generic:less? symbol<?      symbol? symbol?)
(defhandler generic:less? string<?      string? string?)
(defhandler generic:less? (constant #f) null?   null?)
(defhandler generic:less? list<?        list?   list?)
(defhandler generic:less? (lambda (x y) (and (not x) y))                            boolean? boolean?)
(defhandler generic:less? (lambda (x y) (list<? (vector->list x) (vector->list y))) vector?  vector?)
```

Problem 2.6:

The main cost in the predicate-based dispatch mechanism is that each potential procedure match may need to be checked: with n implementations of a particular generic procedure, O(n) will need to be checked before a match is found. With tagged types, the tag tuple determines the procedure to be applied without any computation, and so the dispatch can be done with an O(1) hash table lookup instead of a linear search. If the tags are known at compile-time the dispatch can even be determined with zero runtime cost.

The predicate-based system is extremely flexible; in addition to being dynamic (runtime type definitions are easy, and the definition of a type is itself as flexible as any predicate can be), it can recognize special types at runtime even when inference based solely on types can't determine a special type. For example, in numerical linear algebra there are special routines for operations on symmetric matrices (e.g. in LAPACK) or diagionally dominant matrices, and while some operations always produce such special matrices (so that a tag-based type system could dispatch those special routines), there are common cases where the existence of such structure can only be known at runtime. A predicate-based dispatch system would be able to exploit such structure with no programmer effort.

Since the "overhead" involved in such dispatch mechanisms depends on the number of procedure implementations, and since the relative size of that overhead depends on other features of the computation (such as the time complexity of the dispatched predicates), the "optimal" behavior is very problem-dependent. I think (based on some mailing list post I think I recall) that the 'backslash' operator in Matlab is implemented in tens of thousands of lines of Fortran (perhaps not even counting the LAPACK solver routines it calls to do the "real work") so that special matrix structures can be probed and dispatched upon if the matrix is large enough to justify such effort. That can be very convenient for a high-level programmer, though it's usually easier just to know enough about a numerical computation so that one knows which specific routines to call (without generic dispatch).