MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.945/6.S081—Large-Scale Symbolic Systems
Spring 2014

Pset 2

Issued: 19 February 2014 Due: 26 February 2014

Reading: SICP sections 2.4 and 2.5 (Tagged data, Data-directed programming, Generic Operations)

Code: `ghelper.scm`, `generic-specs.scm`, `generic-sequences.scm`, `load.scm`

Documentation: The MIT/GNU Scheme documentation online at
`http://www.gnu.org/software/mit-scheme/`

One particularly drastic way to extend an existing program to new applications is to extend the operations that it is constructed out of to handle data that the program was not originally expected to operate on. So, for example, a program that was originally intended to compute a numerical result from numerical inputs might be usefully extended to operate on square matrices by extending all of the numerical primitives to do the analogous operations on square matrices as well as numbers. Of course, this is dangerous because the the author of the original program did not know that his code would be extended in this way and may very well have depended on the commutativity of numerical multiplication. On the other hand, the resulting program will still work fine on the original application, and it might just work for the new application. Or it might just take a few minor edits to fix the bugs that arise from the violated assumptions built into the original program. Certainly, the extended program must be tested extensively before we believe that it is correct, and it should be annotated with assertions that test for the preservation of invariants that are expected to hold in the new application.

In this problem set we introduce a very general (and quite expensive) generic operation mechanism that allows us to easily extend the "primitive" operations that a program may depend upon. We will make extensive use of this technique in this class, so it is important to get familiar with its use and misuse.

Fun with Generic Operations

The procedures in the file `ghelper.scm` are an elegant mechanism for implementing generic-operator dispatch, where the handlers for the generic operators are specified by the predicates that the arguments satisfy. The file `generic-specs.scm` is an informal programmer's specification of generic operations that can be defined over a variety of ordered linear data structures, such as lists, vectors, and strings. The file `generic-sequences.scm` is a beginning implementation of the generic operators specified in `generic-specs.scm`.

To make a generic operator with `make-generic-operator` we supply an arity (number of arguments), and optionally a debugging name and a default operation for the operator. So, for example, we might want to define a binary addition operator, with Scheme arithmetic as default:

```
(define add (make-generic-operator 2 'addition +))
```

The arity comes first with `make-generic-operator`. Indeed, its formal parameter list is:

```
(define (make-generic-operator arity #!optional name default)
  ...)
```

Suppose we want `add` to be able to append lists and append strings, as well as add numbers. We can add handlers with `defhandler` as follows:

```
(defhandler add append list? list?)

(defhandler add string-append string? string?)
```

Although we define a few generic sequence operators in `generic-sequences.scm` without specifying the `name` or `default`, it is very useful to at least specify a name to help with debugging. So instead of using our

```
(define sequence:ref (make-generic-operator 2))
```

you should change these to something like

```
(define sequence:ref (make-generic-operator 2 'ref))
```

so you will have less trouble identifying an operator when things go wrong.

To load `ghelper` and `generic-sequences` do `(cd` *`your-code-directory`*`)` and `(load "load")` in a Scheme read-eval-print loop.

Problem 2.1

Complete the implementation started in `generic-sequences.scm` to match the specifications in `generic-specs.scm`. Demonstrate that each of your generic operators works as specified, by showing examples. You should insert your tests as comments in the code you hand in.

Notice that the types in the underlying Scheme are not uniformly specified, so this is not entirely trivial: in our seed file, `generic-sequences.scm`, for example, we had to define `vector-null?`, `list-set!`, and `vector-append` just to fill things out a bit.

Problem 2.2

Operations like `sequence:append` can be extended to allow the combination of unlike sequences. For example, we might expect to be able to write

```
(sequence:append (list 'a 'b 'c) (vector 'd 'e 'f))
```

and get back the list `(a b c d e f)`, assuming that we want a sequence of the first argument type to be the sequence type of the result.

One way to implement this sort of thing is to write specific handlers for all the combinations of types we might want. This may be a large problem. However, the problem can be mitigated by using coercions, such as `vector->list`, `list->vector`, etc. The cost of doing the coercions is the construction of a new intermediate data structure that is not needed in the result. This may or may not be important, depending on the application. With coercions, we make up and use new combinators to help construct the generic operator entries:

```
(define (compose-1st-arg f g)
  (lambda (x y) (f (g x) y)))

(define (compose-2nd-arg f g)
  (lambda (x y) (f x (g y))))
```

Using these we can write such things as:

```
(defhandler generic:binary-append
  (compose-2nd-arg vector-append list->vector)
  vector? list?)

(defhandler generic:binary-append
  (compose-2nd-arg append vector->list)
  list? vector?)
```

Examine the generic specifications. What generalizations that mix combinations of sequence types may be useful? Amend the specification document so as to include the generalization. (Turn in the amended specification sheet with your changes clearly indicated.) Amend your implementation to make these generalizations.

Some of the coercions that you may need are provided by Scheme, but others may need to be written, such as `vector->string`. (Consult the online MIT/GNU Scheme reference manual to see what is and is not provided.)

Problem 2.3

The generic procedure `sequence:append` also illustrates the problem that we must arbitrarily select the type of the output value for each generic procedure. However, the type required may be different depending upon the way that value will be used. For example, complex numbers are best expressed in polar form if we intend to multiply them, whereas they are best expressed in rectangular form if we intend to add them. Automagically knowing the right form is itself an interesting and complicated problem (that we may address later in this subject) but it may be useful, as an intermediate solution, to allow the user to specify the desired output type for each call site. This provides an advantage over explicit after-the-fact coercion of the output by the user, because it gives the dispatch mechanism the ability to select what may be the best way to accomplish the desired result. For example, it may be much more expensive to develop the answer the wrong format than to develop it in the desired format.

A user might specify the target as the type of the result of the `sequence:append` operation by giving an (optional) first argument that is a type predicate as follows:

If a vector is desired the user could specify

```
(sequence:append vector? (list 'a 'b 'c) (vector 'd 'e 'f))
```

if a list is desired the user could specify

```
(sequence:append list? (list 'a 'b 'c) (vector 'd 'e 'f))
```

the system default is specified by omitting the type predicate

```
(sequence:append (list 'a 'b 'c) (vector 'd 'e 'f))
```

Part A: Is this a good idea? (Please state and argue your opinion.) Are there disadvantages to this syntactic scheme? Do you have a better idea?

Part B: What changes would you have to make in the `ghelper.scm` file to implement some form of target selection (either the suggestion above or your better idea)? For example, how would `make-generic-operator` have to change? `defhandler`? Implement your changes and test them.

Problem 2.4

The code for `sequence:append` illustrates another interesting problem. Our generic dispatch program does not allow us to make generic operations with unspecified arity—that take many arguments—such as addition. We programmed around that restriction by defining a binary generic operation and then using a folding reduction, `fold-right`, to extend the binary operation to take an arbitrary number of arguments. However, the folding reduction needs to know the null sequence of the type being constructed. Alternatively, we could have extended the generic dispatch to allow creation of procedures with unspecified arity. This would allow us to move the folding to the type-specific procedures rather than make it a wrapper around the binary generic procedure.

Part A: Is this a good idea? (Please state and argue your opinion.) How does this interact with Problem 2.3 above?

Part B: Assuming that we want to do this, what changes would you have to make in the `ghelper.scm` file? For example, how would `make-generic-operator` have to change? `defhandler?` We do not want you to actually implement these changes, just think about what would have to be done and informally describe your conclusions.

Problem 2.5

Ben Bitdiddle is pleased with our generic sequences but notes that, beyond generic N-tuples, it is useful also to have generic sets. He proposes that we further extend our language with:

```
(generic:sequence->set <sequence>)
```

Which returns a list corresponding to `<sequence>` with no duplicates. Duplication is determined using `EQUAL?` (not `EQ?` nor `EQV?`).

The remaining traditional set operations are straightforward:

```
(set:equal?         <set-1> <set-2>)
(set:union          <set-1> <set-2>)
(set:intersection   <set-1> <set-2>)
(set:difference     <set-1> <set-2>) - E.g. {A,B,C}-{9,B,D}={A,C}
(set:strict-subset? <set-1> <set-2>)
```

Alyssa P. Hacker is quick to point out that an efficient way to implement sets is as sorted, irredundant lists. She adds, "Of course, this would require a `generic:less?` predicate to induce a total order on the potential set elements."

To that end, Alyssa proposes the following ordering on types of objects:

```
null < Boolean < char < number < symbol < string < vector < list
```

She notes that MIT Scheme already provides handy implementations of each of: `char<?`, `<`, `symbol<?`, and `string<?`. Adding that `null<?` and `boolean<?` are straightforward to define and that `vector<?` can just cheat and resort to `list<?` (for now), she cautions that `list<?`, on the other hand, must take special care to ensure that:

```
(generic:less? x y)  implies  (not (generic:less? y x))
```

in order to be well defined (and, thus, well behaved), although `list<?` can, of course, leverage `generic:less?` in any recursive subexpression predications.

Louis Reasoner, ignoring this advice, proposes the following implementation of `list<?`:

```
(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
          ((null? list-1) #f)           ; same
          (else
           (or (generic:less? (car list-1) (car list-2))
               (generic:less? (cdr list-1) (cdr list-2)))))))
```

Alyssa counters that the following is more appropriate:

```
(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)))))))))
```

As a parting shot, Alyssa also advises that entering N^2 items into the generic dispatch table can be avoided by just defining `generic:less?` outright, as per:

```
(define (generic:less? x y)
  (cond ((null?    x) (if (null?    y) (null<?    x y) #t))
        ((null?    y) #f)
        ((boolean? x) (if (boolean? y) (boolean<? x y) #t))
        ((boolean? y) #f)
        ...
        (else (error "Unrecognized data type" x))))
```

Part A: What's wrong with Louis's implementation of the `list<?` predicate? Give a simple example and a brief explanation of what problems this would cause if it were used in `generic:less?` to sort sets.

Part B: Briefly critique Alyssa's suggesting for implementing `generic:less?` as an explicit case analysis versus using the dispatch table.

Part C: Implement and demonstrate a `generic:less?` operation using Alyssa's total ordering of data types (and her `list<?` code), but using the generic dispatch mechanism instead of an explicit conditional shown above.

Problem 2.6

The system for implementing generic operations that we have looked at so far in this problem set is extremely general and flexible: the dispatch to a handler is based on arbitrary predicates applied to the arguments. Most generic operation systems are more constrained, in that the arguments are presumed to have types that are determined either statically by some declaration mechanism or by a type tag that is associated with the argument data. For example, in the SICP readings for this problem set, the data is tagged and the dispatch is based on these tags. Such a tagged-data system has important advantages of efficiency, but it gives up some flexibility.

How much does dispatch on predicates cost? What is the fundamental efficiency problem here? Imagine that we have a system with tagged data, but that we test for the tags with predicates. What can be done with the data tags that can eliminate much of the work of the predicate-based system?

On the other hand, what do we give up in a more conventional system, such as the one outlined in SICP, by contrast to the predicate-based system? What is an example of lost flexibility?

Under what circumstances could you build a generic dispatch system with zero runtime overhead? What flexibility would you be giving up? What about a system with very little or constant-time overhead?

Write a few clear paragraphs expounding on these ideas. Try to separate accident from essence. (Some aspects of a system are consequences of accidental choices—ones that could easily be changed—such as the use of a hash table rather than an association list. Other aspects are essential in that no local modifications can significantly change the behavior.)

Problem 2.7

MIT/GNU Scheme supports *streams*, which can have infinite length, like clojure's lazy-seqs or python's sequences. They are part of MIT scheme, and you can see the source to them in the file

`mit-gnu-streams.scm`. Don't load this file—it is already in the system. There is an extensive discussion of streams in section 3.5 of SICP.

Here is an example of a stream of infinite length:

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

(define integers (integers-starting-from 1))
```

You can get as many natural numbers as you want:

```
(stream-head integers 5)
;Value: (1 2 3 4 5)
```

And here is a stream of all the even natural numbers:

```
(define evens (stream-filter even? integers))

(stream->head evens 20)
;Value: (2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32 34 36 38 40)
```

Here is a non-infinite stream.

```
(define finite-stream (list->stream '(1 2 3 4 5)))
;Value: (1 . #[promise 13])
```

Part A: Integrate streams into your generic system where appropriate. Give a few examples of typical usage including infinite and non infinite streams.

Part B: How difficult was it to integrate stream operations? What does this say about the generality of the generic operations system?

For extra fun: MIT/GNU Scheme streams are "odd" streams, which are subtly inferior to "even" streams, as defined in SRFI 41 (`http://srfi.schemers.org/srfi-41/srfi-41.html`). SRFIs are a set of curated extensions to the scheme language which scheme implementers are free to include in their own implementations of scheme.

Look at SRFI 41: What is the difference between "even" and "odd" streams? Why are "even" streams better?