SICP section 2.5.2

September 16, 2007 at 06:09 Tags SICP

The code for this section is in Scheme.

A short review of the material

This section is about combining data of different types. This is where the generic type system we're developing will really shine. But before we delve deeper into the code, it is beneficial to first gain a more thorough understanding of how we implement tagging.

Like many things in Lisp, lists can be used to simply represent tagged types. Attaching a tag type is just prepending its name to the datum:

```
(define (attach-tag type-tag contents)
  (if (number? contents)
    contents
    (cons type-tag contents)))
```

The code above includes an optimization for scheme numbers, but other than that, it's a simple `cons`.

We even saw how to use this simple technique to build hierarchical tag systems. In this case the attached tags act in a LIFO manner – the higher-level tags ("a complex number") come before the lower-level tags ("a rectangular implementation of a complex number").

The key function in all of this is `apply-generic`, and it's worth understanding it thoroughly:

```
(define (apply-generic op . args)
  (let ((type-tags (map type-tag args)))
    (let ((proc (get op type-tags)))
      (if proc
          (apply proc (map contents args))
          (error
            "No method for these types -- APPLY-GENERIC"
            (list op type-tags))))))
```

Recall that our operation table (`get` and `put`) dispatches to functions based on the requested operation and the types of the arguments. The authors thoughtfully included provisions for multiple arguments of different types in the previous section – to use it fully only here.

`apply-generic` takes an operation (say, `add`) and a list of arguments – normal Lisp variables. It queries the arguments for their types, using the generic `type-tag` accessor, builds a list of types (with `map`) and passes it together with the operation to `get`. This way it gets the correct function for the requested operation and types.

Keep this point in mind – it will be crucial in the understanding of the code of this section. The whole type tagging and operation dispatching thing is implemented using the familiar abstraction of lists and accessors to list elements. There's a small detour into more advanced data structures to implement `put` and `get` efficiently[1], but that could have easily been done with lists too. There's nothing magical and special about it – we've been using list abstractions from the very start of this book, and this topic should be very familiar by now.

Coercion

Here's the code that implements the coercion table. It is very similar to the operator dispatch table:

```
(define *coercion-table* (make-hash-table 'equal))

(define (put-coercion type-from type-to proc)
  (hash-table-put!
    *coercion-table*
    (list type-from type-to)
    proc))

(define (get-coercion type-from type-to)
  (hash-table-get
    *coercion-table*
    (list type-from type-to)
    #f))
```

And the `apply-generic` that uses it is:

```
(define (apply-generic op . args)
  (let ((type-tags (map type-tag args)))
    (let ((proc (get op type-tags)))
      (if proc
          (apply proc (map contents args))
          (if (= (length args) 2)
              (let ((type1 (car type-tags))
                    (type2 (cadr type-tags))
                    (a1 (car args))
                    (a2 (cadr args)))
                (let ((t1->t2 (get-coercion type1 type2))
                      (t2->t1 (get-coercion type2 type1)))
                  (cond (t1->t2
                         (apply-generic op (t1->t2 a1) a2))
                        (t2->t1
                         (apply-generic op a1 (t2->t1 a2)))
                        (else
                         (error "No method for these types"
                                (list op type-tags))))))
              (error "No method for these types"
                     (list op type-tags)))))))
```

Exercise 2.81

a. Louis's coercion procedures don't help much. It becomes very obvious if you study the code of `apply-generic` – it calls itself recursively on coerced types, and hence for numbers it doesn't add any value. For complex numbers (or any other numbers for which the requested operation is not implemented) it's even worse – `apply-generic` gets into an infinite recursion[1].

b. Louis is wrong. `apply-generic` works just fine as-is. If two arguments are of the same type and `apply-generic` couldn't find an operation for them with `get`, self coercion won't help.

c. Here's `apply-generic` that doesn't try coercion for two arguments of the same type:

```
(define (apply-generic op . args)
  (define (no-method type-tags)
    (error "No method for these types"
      (list op type-tags)))

  (let ((type-tags (map type-tag args)))
    (let ((proc (get op type-tags)))
      (if proc
          (apply proc (map contents args))
          (if (= (length args) 2)
              (let ((type1 (car type-tags))
                    (type2 (cadr type-tags))
                    (a1 (car args))
                    (a2 (cadr args)))
                (if (equal? type1 type2)
                  (no-method type-tags)
                  (let ((t1->t2 (get-coercion type1 type2))
                        (t2->t1 (get-coercion type2 type1))
                        (a1 (car args))
                        (a2 (cadr args)))
                    (cond (t1->t2
                           (apply-generic op (t1->t2 a1) a2))
                          (t2->t1
                           (apply-generic op a1 (t2->t1 a2)))
                          (else (no-method type-tags))))))
              (no-method type-tags))))))
```

Note that I defined `no-method` to avoid repetitive code.

Exercise 2.82

This strategy requires a much more complicated `apply-generic`. We can contain the complexity by defining a few internal functions that give convenient names to processes. Now the main flow of `apply-generic` should not be too hard to understand.

```
(define (apply-generic op . args)
  (define (can-coerce-into? types target-type)
    "Can all _types_ be coerced into _target-type_ ?"
    (andmap
      (lambda (type)
        (or
          (equal? type target-type)
          (get-coercion type target-type)))
      types))
  (define (find-coercion-target types)
    "Find a type among _types_ that all _types_ can be
    coerced into."
    (ormap
      (lambda (target-type)
        (if (can-coerce-into? types target-type)
          target-type
          #f))
      types))
  (define (coerce-all args target-type)
    "Coerce all _args_ to _target-type_"
    (map
      (lambda (arg)
        (let ((arg-type (type-tag arg)))
          (if (equal? arg-type target-type)
            arg
            ((get-coercion arg-type target-type) arg))))
      args))
  (define (no-method type-tags)
    (error "No method for these types"
      (list op type-tags)))
  (let ((type-tags (map type-tag args)))
    (let ((proc (get op type-tags)))
      (if proc
        (apply proc (map contents args))
        (let ((target-type (find-coercion-target type-tags)))
          (if target-type
            (apply
              apply-generic
              (append
                (list op)
                (coerce-all args target-type)))
            (no-method type-tags)))))))
```

A situation where this strategy is not sufficiently general: we are trying to coerce the arguments only to types that are present in the call, and so can miss other types.

Consider the example in Figure 2.26 in the book. Suppose we have a call with these types: `(kite quadrilateral)`. Since `kite` can be coerced into `quadrilateral`, everything works as expected.

But now suppose we have a call with: `(triangle kite quadrilateral)`. Going over each of these types and trying to coerce them to each other won't work. But they *can* all be coerced into `polygon`. This demonstrates the flaw of this method. What we should be really going is finding some common “ancestor” type for all the types we work on.

Exercise 2.83

```
;; Into integer package
(define (integer->rational n)
  (make-rational n 1))

(put 'raise '(integer)
  (lambda (i) (integer->rational i)))

;; Into rational package
(define (rational->real r)
  (make-real
    (exact->inexact
      (/ (numer r) (denom r)))))

(put 'raise '(rational)
  (lambda (r) (rational->real r)))

;; Into real package
(define (real->complex r)
  (make-complex-from-real-imag r 0))

(put 'raise '(real)
  (lambda (r) (real->complex r)))

(define (raise x)
  (apply-generic 'raise x))
```

Exercise 2.84

For simplicity’s sake, I’m going to use the `apply-generic` that only works with two arguments. Also, I’ll employ only the types we’re familiar with: scheme number, rational, complex.

```
(define (apply-generic-r op . args)
  (define (no-method type-tags)
    (error "No method for these types"
      (list op type-tags)))
  (define (raise-into s t)
    "Tries to raise s into the type of t. On success,
    returns the raised s. Otherwise, returns #f"
    (let ((s-type (type-tag s))
          (t-type (type-tag t)))
      (cond
        ((equal? s-type t-type) s)
        ((get 'raise (list s-type))
          (raise-into ((get 'raise (list s-type)) (contents s)) t))
        (t #f))))

  (let ((type-tags (map type-tag args)))
    (let ((proc (get op type-tags)))
      (if proc
          (apply proc (map contents args))
          (if (= (length args) 2)
            (let ((o1 (car args))
                  (o2 (cadr args)))
              (cond
                ((raise-into o1 o2)
                  (apply-generic-r op (raise-into o1 o2) o2))
                ((raise-into o2 o1)
                  (apply-generic-r op o2 (raise-into o2 o1)))
                (t (no-method type-tags))))
            (no-method type-tags))))))
```

The auxiliary function `raise-into` plays a key role here. It tries to recursively raise its first argument into the type of the second argument. If successful, it returns the first argument with the type of the second. Otherwise it returns `#f`.

Although this method is compatible with the rest of the system and imposes the minimal overhead to the action of adding new types, it is not very efficient. In a real system, it probably would have made sense to precompute the relationships between types and not try to figure it out every time anew.

Exercise 2.85

Let's begin with the `project` functions for the various types:

```scheme
(put 'project '(rational)
  (lambda (r)
    (make-scheme-number
      (floor (/ (numer r) (denom r))))))

(put 'project '(real)
  (lambda (r)
    (let ((scheme-rat
            (rationalize
              (inexact->exact r) 1/100)))
      (make-rational
        (numerator scheme-rat)
        (denominator scheme-rat)))))

(put 'project '(complex)
  (lambda (c) (make-real (real-part c))))
```

The functions `floor`, `rationalize` and `inexact->exact` are standard Scheme functions for dealing with conversions between different numbers. The functions `numerator` and `denominator` are Scheme's own accessors to rational numbers. It was convenient to use them together with the output of `rationalize`.

Here is `drop` and a modified `apply-generic-r`:

```
(define (drop num)
  (let ((project-proc
         (get 'project (list (type-tag num)))))
    (if project-proc
      (let ((dropped (project-proc (contents num))))
        (if (equ? num (raise dropped))
          (drop dropped)
          num))
      num)))

(define (apply-generic-r op . args)
  (define (no-method type-tags)
    (error "No method for these types"
      (list op type-tags)))
  (define (raise-into s t)
    "Tries to raise s into the type of t. On success,
    returns the raised s. Otherwise, returns #f"
    (let ((s-type (type-tag s))
          (t-type (type-tag t)))
      (cond
        ((equal? s-type t-type) s)
        ((get 'raise (list s-type))
          (raise-into ((get 'raise (list s-type)) (contents s)) t))
        (t #f))))

  (let ((type-tags (map type-tag args)))
    (let ((proc (get op type-tags)))
      (if proc
          (drop (apply proc (map contents args)))
          (if (= (length args) 2)
            (let ((o1 (car args))
                  (o2 (cadr args)))
              (cond
                ((raise-into o1 o2)
                  (apply-generic-r op (raise-into o1 o2) o2))
                ((raise-into o2 o1)
                  (apply-generic-r op o2 (raise-into o2 o1)))
                (t (no-method type-tags))))
            (no-method type-tags))))))
```

Note that the only change `apply-generic-r` has undergone is the call to `drop` on `apply` – which is the exit point, in order to simplify the answer.

The way I decided to implement `drop` allows, IMHO, for the most generic type system. `drop` tests if the type it's given has a `project` function, so new types without `project` will also work as expected.

Exercise 2.86

We should first think about the abstraction level on which we'd want to implement this feature. On the level of `complex`, only accessors are used (`real-part`, `angle` etc.) so if we implement those, everything will work as expected. So, we should operate on the level of `rectangular` and `polar`.

What is needed is pretty simple; we can just make the operations:`square`, `arctan`, `sine` and `cosine` generic procedures, and implement them for each number system we're installing. Then, the accessors of `rectangular` and `polar` will just use these generic procedures without worrying about the internals. This way we also ensure that when new types are added into the system, they can be easily be incorporated as the elements of complex numbers, by implementing the appropriate generic procedures.

[1] Recall that we used Scheme's hash tables for that.

[2] It's a curious fact to note that the Scheme interpreter (I'm usingPLT's MzScheme) doesn't throw a stack overflow error, but dutifully engages in an infinite loop. This is because the recursive call of `apply-generic` is a *tail call*, and the Scheme interpreter automatically optimizes it into a loop. Contrast this with the CL runtime we were using in other sections – while it implements the tail call optimization for compiled code, it doesn't do it for interpreted code, so it throws a stack overflow.

For comments, please send me an email.