

GENERIC OPERATORS 9

GEORGE WANG
gswang.cs61a@gmail.com
Department of Electrical Engineering and Computer Sciences
University of California, Berkeley

July 6, 2010

1 Material from Last Week

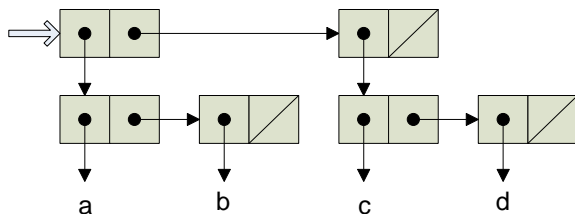
1.1 About Homework/Projects

Many people commented on the length of the projects. However, an equally large number commented on the need for practice problems as well as the help that the homework provided in terms of being able to understand the material. I do admit that I think this homework was fairly sizable in length, but I don't know that it was unreasonable, given that I believe many of the questions should not have taken terribly long.

If you find yourself working on a single problem for more than 10 minutes * the number of stars, please consider working in a group or finding some TA help.

1.2 car/cdr recursion

Consider the deep list $((a\ b)\ (c\ d))$. Ordinarily we would draw its box and pointer diagram with a horizontal spine at the top and the sublists beneath the spine:



But imagine that we grab the first pair of this structure and “shake” it so that the pairs fall down as far as they can. We’d end up with this diagram:

Just to get away from the arithmetic examples in the text, we'll use another example about geometric shapes. Our data types will be squares and circles; our operations will be area and perimeter. We want to be able to say, e.g., `(area circle3)` to get the area of a particular (previously defined) circle. To make this work, the function `area` has to be able to tell which type of shape it's seeing. We accomplish this by attaching a type tag to each shape:

```
;;;;; In file cs61a/lectures/2.4/geom.scm
(define pi 3.141592654)
(define (make-square side) (attach-tag 'square side))
(define (make-circle radius) (attach-tag 'circle radius))
(define (area shape)
  (cond ((eq? (type-tag shape) 'square)
        (* (contents shape) (contents shape)))
        ((eq? (type-tag shape) 'circle)
         (* pi (contents shape) (contents shape)))
        (else (error "Unknown shape -- AREA"))))
(define (perimeter shape)
  (cond ((eq? (type-tag shape) 'square)
        (* 4 (contents shape)))
        ((eq? (type-tag shape) 'circle)
         (* 2 pi (contents shape)))
        (else (error "Unknown shape -- PERIMETER"))))
;; some sample data
(define square5 (make-square 5))
(define circle3 (make-circle 3))
```

4 Orthogonality of Types and Operators

The title, orthogonality, means that all the functions and all the operators should be combinable. Thus, you can call every function on every type of data.

The next problem to deal with is the proliferation of functions because you want to be able to apply every operation to every type. In our example, with two types and two operations we need four algorithms. What happens when we invent a new type? If we write our program in the conventional (i.e., old-fashioned) style as above, it's not enough to add new functions; we have to modify all the operator functions like `area` to know about the new type. We'll look at two different approaches to organizing things better: data-directed programming and message passing.

5 Data Directed Programming

The idea in DDP is that instead of keeping the information about types versus operators inside functions, as `cond` clauses, we record this information in a data structure. A&S provide tools `put` to set up the data structure and `get` to examine it:

```
> (get 'foo 'baz)
#f
> (put 'foo 'baz 'hello)
> (get 'foo 'baz)
hello
```

Once you put something in the table, it stays there. (This is our first departure from functional programming. But our intent is to set up the table at the beginning of the computation and then to treat it as constant information, not as something that might be different the next time you call `get`, despite the example above.) For now we take `put` and `get` as primitives; we'll see how to build them in section 3.3 in three weeks. The code is mostly unchanged from the conventional version; the tagged data ADT and the two shape ADTs are unchanged. What's different is how we represent the four algorithms for applying some operator to some type:

```
;;;;; In file cs61a/lectures/2.4/geom.scm
(put 'square 'area (lambda (s) (* s s)))
(put 'circle 'area (lambda (r) (* pi r r)))
(put 'square 'perimeter (lambda (s) (* 4 s)))
(put 'circle 'perimeter (lambda (r) (* 2 pi r)))
```

Notice that the entry in each cell of the table is a function, not a symbol. We can now redefine the six generic operators (“generic” because they work for any of the types):

```
;;;;; In file cs61a/lectures/2.4/geom.scm
(define (area shape) (operate 'area shape))
(define (perimeter shape) (operate 'perimeter shape))
(define (operate op obj) ;; like APPLY-GENERIC but for one operand
  (let ((proc (get (type-tag obj) op)))
    (if proc
        (proc (contents obj))
        (error "Unknown operator for type"))))
```

Now if we want to invent a new type, all we have to do is a few `put` instructions and the generic operators just automatically work with the new type.

Don't get the idea that DDP just means a two-dimensional table of operator and type names! DDP is a very general, great idea. It means putting the details of a system into data, rather than into programs, so you can write general programs instead of very specific ones. In the old days, every time a company got a computer they had to hire a bunch of programmers to write things like payroll programs for them. They couldn't just use someone else's program because the details would be different, e.g., how many digits in the employee number. These days you have general business packages and each company can “tune” the program to their specific purpose with a data file.

Another example showing the generality of DDP is the compiler compiler. It used to be that if you wanted to invent a new programming language you had to start from scratch in writing a compiler for it. But now we have formal notations for expressing the syntax of the language. (See section 7.1, page 38, of the Scheme Report at the back of the course reader.) A single program can read these formal descriptions and compile any language. [The Scheme BNF is in `~cs61a/lectures/2.4/bnf.`]

6 Message Passing

In conventional style, the operators are represented as functions that know about the different types; the types themselves are just data. In DDP, the operators and types are all data, and there is one universal `operate` function that does the work. We can also stand conventional style on its head, representing the types as functions and the operations as mere data. In fact, not only are the types functions, but so are the individual data themselves. That is, there is a function (`make-circle` below) that represents the circle type, and when you invoke that function, it returns a function that represents the particular circle you give it as its argument. Each circle is an object and the function that represents it is a dispatch procedure that takes as its argument a message saying which operation to perform.

```

;;;;; In file cs61a/lectures/2.4/geom.scm
(define (make-square side)
  (lambda (message)
    (cond ((eq? message 'area) (* side side))
          ((eq? message 'perimeter) (* 4 side))
          (else (error "Unknown message")))))
(define (make-circle radius)
  (lambda (message)
    (cond ((eq? message 'area) (* pi radius radius))
          ((eq? message 'perimeter) (* 2 pi radius))
          (else (error "Unknown message")))))
(define square5 (make-square 5))
(define circle3 (make-circle 3))

```

The defines that produce the individual shapes look no different from before, but the results are different: Each shape is a function, not a list structure. So to get the area of the shape `circle3` we invoke that shape with the proper message: `(circle3 'area)`. That notation is a little awkward so we provide a little “syntactic sugar” that allows us to say `(area circle3)` as in the past:

```

;;;;; In file cs61a/lectures/2.4/msg.scm
(define (operate op obj) (obj op))
(define (area shape) (operate 'area shape))
(define (perimeter shape) (operate 'perimeter shape))

```

Message passing may seem like an overly complicated way to handle this problem of shapes, but we’ll see next week that it’s one of the key ideas in creating object-oriented programming. Message passing becomes much more powerful when combined with the idea of local state that we’ll learn next week. We seem to have abandoned tagged data; every shape type is just some function, and it’s hard to tell which type of shape a given function represents. We could combine message passing with tagged data, if desired, by adding a type message that each object understands.

```

(define (make-square side)
  (lambda (message)
    (cond ((eq? message 'area) (* side side))
          ((eq? message 'perimeter) (* 4 side))
          ((EQ? MESSAGE 'TYPE) 'SQUARE)
          (else (error "Unknown message")))))

```

7 Organization

	○	□
area	(lambda (data) (* pi data data))	(lambda (data) (* data data))
perimeter	(lambda (data) (*2 pi data))	(lambda (data) (* 4 data))

In this lecture, we’ve talked about 3 ways to resolve the problem of having many functions that take in arguments. The first way, which we call conventional style, has functions as rows in this table. This makes adding new data types very difficult. The second way, DDP, involves using this table directly and adding entries. This is what looks like the best way for now. The third way, message passing, involves making the columns the data. In other words, the data is ‘smart’. In that style, it’s very difficult to add new functions.

Message passing should remind you of Object-Oriented Programming, if you’ve seen that before. We’ll see next week that combined with other aspects of OOP, the message passing style is able to overcome many of its weaknesses.