CS 61A Lecture Notes Second Half of Week 3

Topic: Representing abstract data

Reading: Abelson & Sussman, Sections 2.4 through 2.5.2 (pages 169–200)

The overall problem we're addressing in the next two lectures is to control the complexity of large systems with many small procedures that handle several types of data. We are building toward the idea of *object-oriented programming*, which many people see as the ultimate solution to this problem, and which we discuss for two weeks starting next week.

Big ideas:

- tagged data
- data-directed programming
- message passing

The first problem is keeping track of types of data. If we see a pair whose `car` is 3 and whose `cdr` is 4, does that represent $\frac{3}{4}$ or does it represent $3 + 4i$?

The solution is *tagged data*: Each datum carries around its own type information. In effect we do `(cons 'rational (cons 3 4))` for the rational number $\frac{3}{4}$, although of course we use an ADT.

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 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))
```

• Orthogonality of types and operators.

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*.

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 two 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`.]

• 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 `define`s 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")))))
```

• Dyadic operations.

Our shape example is easier than the arithmetic example in the book because our operations only require one operand, not two. For arithmetic operations like `+`, it's not good enough to connect the operation with a type; the two operands might have two different types. What should you do if you have to add a rational number to a complex number?

There is no perfect solution to this problem. For the particular case of arithmetic, we're lucky in that the different types form a sequence of larger and larger sets. Every integer is a rational number; every rational is a real; every real is a complex. So we can deal with type mismatch by *raising* the less-complicated operand to the type of the other one. To add a rational number to a complex number, raise the rational number to complex and then you're left with the problem of adding two complex numbers. So we only need N addition algorithms, not N^2 algorithms, where N is the number of types.

Do we need N^2 raising algorithms? No, because we don't have to know directly how to raise a rational number to complex. We can raise the rational number to the next higher type (real), and then raise that real number to complex. So if we want to add $\frac{1}{3}$ and $2 + 5i$ the answer comes out $2.3333 + 5i$.

As this example shows, nonchalant raising can lose information. It would be better, perhaps, if we could get the answer $\frac{7}{3} + 5i$ instead of the decimal approximation. Numbers are a rat's nest full of traps for the unwary. You will live longer if you only write programs about integers.