CS403: Programming Languages

Notes on Tagged Data

Tagged data

Consider developing a complex number package. Remember there are two common representations of a complex number, polar and rectangular. How do we distinguish the two representations? For example, does the list `(2 3)` represent 2+3i (rectangular form) or 2r3 (polar form)? We can clarify the situation by using a symbol in the list (the symbol is known as a tag). For example `(Rectangular 2 3)` would refer to 2+3i while `(Polar 2 3)` would refer to 2r3.

Here's a definition of two constructors which apply the appropriate tag.

```
(define (makeComplexR x y)
    (list 'Rectangular x y)
    )
(define (makeComplexP x y)
    (list 'Polar x y)
    )
```

We will also need selectors, to access the type (representation) and the contents of a complex number.

```
(define (type a)
    (car a)
    )
(define (contents a)
    (cdr a)
    )
```

Suppose I wish to add two complex numbers together. I could define four routines

- addComplexRR
- addComplexRP
- addComplexPR
- addComplexPP

to handle each possible combination of representations. The first routine, for example, would expect both arguments in rectangular form while the third routine would expect a polar representation for the first argument and a rectangular representation for the second.

Yes, I could define four routines but I would adding to the ugliness in the world. Better would be to make the addition function independent of the representation. I could implement *addComplex* as a case statement which performed the proper calculations based upon the types of the arguments (I would need four cases) but this is little better than the first attempt at a solution. The fact that addition is much easier if the complex numbers are in rectangular form suggests a nice solution. We will just assume that the arguments are in rectangular form. Here's a definition given that assumption.

```
(define (addComplex a b)
    (makeComplexR
        (+ (real a) (real b))
        (+ (imag a) (imag b))
        )
    )
```

If we design all our functions independent of the representation, then the fact that we chose to represent the resulting complex number in rectangular form will matter not a whit.

Now we can define the selectors *real* and *imag*, which will pull out the real part of the number if it is in rectangular form and calculate the real part if it is in polar form.

```
(define (real a)
    (if (eq? (type a) 'Polar)
        (* (car (contents a)) (cos (cadr (contents a))))
        (car (contents a))
        )
    )
(define (imag a)
    (if (eq? (type a) 'Polar)
        (* (car (contents a)) (sin (cadr (contents a))))
        (cadr (contents a))
        )
    )
```

While this solution is a good one, what happens if we wish to add more representations of complex numbers? Our implementations of *real* and *imag* would need to be modified to add the new representations. While this is not an onerous task, there is a better way through data-directed programming.

Data directed programming

Data-directed programming generally involves a table of functions which is (partially) indexed by data types. The proper function to be used to operate on the data is retrieved from the table based upon the types of the data to be operated upon. As the book claims, this is indeed a powerful way to program. Here is a data directed solution to the problem brought up at the end of the previous section.

First we postulate the existence of a table to store these functions. We will use the function *put* to place functions into the table and the function *get* to retrieve functions. Here are the calls to *put* which place different versions of the real selector into the table

```
    (put
        'real
        '(Rectangular)
        (lambda (x) (car x))
        )
    (put
        'real
        '(Polar)
        (lambda (x) (* (car x) (cos (cadr x))))
        )
```

Now we can redefine the real selector so that it calls a helper function which extracts the proper version of the selector out of the table and applies that function to the number.

```
    (define (real a)
        (apply-generic 'real a)
        )
    (define (imag a)
        (apply-generic 'imag a)
        )
```

The fact that the call to *put* (to install the *real* selector) had `'real` as the first argument and the helper function *apply-generic* also had `'real` as its first argument is no accident. The table is indexed by the generic operator name (in this case `'real`) and a list of argument types. In this case the selector takes but a single argument. Hence the single type in the list passed as the second argument to *put*. The helper function *apply-generic* will retrieve the proper function using `'real` and a list of types derived from the remaining arguments. Here is the definition of *apply-generic*.

```
    (define (apply-generic op . args)
        (let*
            ((types (map type args))
             (f (get op types)))
            (apply f (map contents args))
            )
        )
```

The first variable in the *let** is *types* and it is set to list of argument types via *map*. Once that list is constructed, the second variable *f* is set the function that was previously installed in the table. In the body of the *let**, that function is applied to the contents part of the argument list.

Now lets go back to the selector real. Suppose we make the following definitions

```
(define a (makeComplexR 3 4))
(define b (makeComplexP 5 6))
```

and evaluate the following expressions

```
(real a)
```

Using substitution, we get

```
(apply-generic real (Rectangular 3 4))
```

Within *apply-generic*, *op* is bound to the symbol `real` and *args* is bound to the list (Rectangular 3 4). The variable *types* is then bound to the list (Rectangular) and the *f* is bound the result of the call to *get*, which resolves to

```
(get real (Rectangular))
```

The result of this binding gives *f* a value of

```
(lambda (x) (car x))
```

Next the *apply* function is called as

```
(apply (lambda (x) (car x)) ((3 4)))
```

This is converted by the interpreter to

```
((lambda (x) (car x)) 3 4)
```

which returns

```
3
```

which is the real part of *a*. Whew! Evaluating

```
(real b)
```

follows a similar path. Now imagine what happens if we add a new representation. All we need to do is define a new constructor (with an appropriate tag) and install the proper functions into the table. That's it. None of the other code needs to be changed at all!