MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.001—Structure and Interpretation of Computer Programs
Fall Semester, 1997

Lecture Notes – Oct. 13, 1997

Generic Operations

We are going to build a system for arithmetic manipulation of numbers.

Operators for manipulating complex numbers

```
(define (+c z1 z2)
  (make-rectangular (+ (real-part z1) (real-part z2))
                    (+ (imag-part z1) (imag-part z2))))

(define (-c z1 z2)
  (make-rectangular (- (real-part z1) (real-part z2))
                    (- (imag-part z1) (imag-part z2))))

(define (*c z1 z2)
  (make-polar (* (magnitude z1) (magnitude z2))
              (+ (angle z1) (angle z2))))

(define (/c z1 z2)
  (make-polar (/ (magnitude z1) (magnitude z2))
              (- (angle z1) (angle z2))))
```

One way to represent complex numbers is in rectangular form, where a complex number data object is represented by a cons cell of the real and imaginary parts.

The specific implementation is:

```
(define (make-rectangular x y) (cons x y))

(define (real-part z) (car z))
(define (imag-part z) (cdr z))
(define (make-polar r a) (cons (* r (cos a)) (* r (sin a))))
(define (magnitude z) (sqrt (+ (square (car z)) (square (cdr z)))))
(define (angle z) (atan (cdr z) (car z)))
```

On the other hand, we may choose to implement our complex numbers in polar form as as a cons cell of the magnitude and angle.

```
(define (make-rectangular x y)
  (cons (sqrt (+ (square x) (square y)))
        (atan y x)))

(define (real-part z) (* (car z) (cos (cdr z))))
(define (imag-part z) (* (car z) (sin (cdr z))))
(define (make-polar r a) (cons r a))
(define (magnitude z) (car z))
(define (angle z) (cdr z))
```

In order to support both types of complex numbers we need to be able to guarantee that, for instance, `real-part` always returns the real part of a complex number, independent of whether it is represented in rectangular or polar coordinates.

In order to do this we need TYPED data objects, where the types can be seen by inspection.

Building typed data objects:

```
(define (attach-type type contents) (cons type contents))

(define (type datum)
  (if (pair? datum)
      (car datum)
      (error "Bad typed datum -- TYPE" datum)))

(define (contents datum)
  (if (pair? datum)
      (cdr datum)
      (error "Bad typed datum -- CONTENTS" datum)))
```

We can identify different kinds of complex numbers by examining their types:

```
(define (rectangular? z) (eq? (type z) 'rectangular))

(define (polar? z) (eq? (type z) 'polar))
```

To make this work, we must add on a type whenever we construct a complex number in the rectangular representation.

```
(define (make-rectangular x y)
  (attach-type 'rectangular (cons x y)))
```

Here is the rest of the rectangular package. It is the same as before, except we have changed the names of the selectors so as to avoid name conflicts with the polar pakage:

```
(define (real-part-rectangular z) (car z))

(define (imag-part-rectangular z) (cdr z))

(define (magnitude-rectangular z)
  (sqrt (+ (square (car z))
           (square (cdr z)))))

(define (angle-rectangular z) (atan (cdr z) (car z)))
```

The polar package is handled analogously:

```
(define (make-polar r a) (attach-type 'polar (cons r a)))
(define (real-part-polar z) (* (car z) (cos (cdr z))))
(define (imag-part-polar z) (* (car z) (sin (cdr z))))
(define (magnitude-polar z) (car z))
(define (angle-polar z) (cdr z))
```

Now we can define the generic selectors for complex numbers:

```
(define (real-part z)
  (cond ((rectangular? z)
         (real-part-rectangular (contents z)))
        ((polar? z)
         (real-part-polar (contents z)))))

(define (imag-part z)
  (cond ((rectangular? z)
         (imag-part-rectangular (contents z)))
        ((polar? z)
         (imag-part-polar (contents z)))))

...
```

Now suppose we want to allow for other kinds of numbers in our system.

For example, we have ordinary numbers:

```
(define (make-scheme-number n)
  (attach-tag 'scheme-number n))

(define (scheme-number? n)
  (eq? (type-tag n) 'scheme-number))

(define (add-scheme-number n1 n2)
  (make-scheme-number (+ n1 n2)))
```

and ditto for other operations, like `mul, sub, div`.

We have similar tagging and operations on other types of numbers, such as rationals.

Now we need a generic add operation, utilizing dispatch on type:

```
(define (add x y)
  (cond ((and (complex? x) (complex? y))
         (add-generic-complex (contents x) (contents y)))
        ((and (rational? x) (rational? y))
         (add-rat (contents x) (contents y)))
        ((and (scheme-number? x) (scheme-number? y))
         (add-scheme-number (contents x) (contents y)))))
```

To really make generic operations, though, we need to utilize generic calls to our operations inside the individual procedures, for example:

```
(define (add-complex z1 z2)
  (make-from-real-imag (ADD (real-part z1) (real-part z2))
                       (ADD (imag-part z1) (imag-part z2))))

(define (add-rat x y)
  (make-rat (ADD (MUL (numer x) (denom y))
                 (MUL (numer y) (denom x)))
            (MUL (denom x) (denom y))))
```

Tracing through an example will illustrate to you how the generic calls to ADD, MUL, etc. allow one to automatically extend our system to deal with arbitrarily complex data types.

To support this generality, we need to build good interfaces to the generic operators. We have already seen one example, which is the notion of "dispatch on type" in which each operator uses the tag information attached to its input arguments to determine which specific operation to use.

A more general method would build a table relating operations on the one hand with argument types on the other. Assume that we have two operations associated with such a table:

- `(put` op type item) installs the item in the table, indexed by the op and the type.
- `(get` op type) looks up the op, type entry in the table and returns the item found there. If no item is found, `get` returns false.

Now we can install our procedures directly into the table:

```
(define (install-rectangular-package)
  ;; internal procedures
  (define (real-part z) (car z))
  (define (imag-part z) (cdr z))
  (define (make-from-real-imag x y) (cons x y))
  (define (magnitude z)
    (sqrt (+ (square (real-part z))
             (square (imag-part z)))))
  (define (angle z)
    (atan (imag-part z) (real-part z)))
  (define (make-from-mag-ang r a)
    (cons (* r (cos a)) (* r (sin a))))

  ;; interface to the rest of the system
  (define (tag x) (attach-tag 'rectangular x))
  (put 'real-part '(rectangular) real-part)
  (put 'imag-part '(rectangular) imag-part)
  (put 'magnitude '(rectangular) magnitude)
  (put 'angle '(rectangular) angle)
  (put 'make-from-real-imag 'rectangular
     (lambda (x y) (tag (make-from-real-imag x y))))
  (put 'make-from-mag-ang 'rectangular
     (lambda (r a) (tag (make-from-mag-ang r a))))
  'done)
```

and similarly for the polar package of complex numbers.

To use these tables we have:

```
(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  -- APPLY-GENERIC"
            (list op type-tags))))))
```

Using `apply-generic`, we can define our generic selectors as follows:

```
(define (real-part z)
   (apply-generic 'real-part z))

(define (imag-part z)
   (apply-generic 'imag-part z))

(define (magnitude z)
   (apply-generic 'magnitude z))

(define (angle z)
   (apply-generic 'angle z))
```

We can export the constructors outside of the package, so that other procedures can use them:

```
(define (make-from-real-imag x y)
  ((get 'make-from-real-imag 'rectangular) x y))

(define (make-from-mag-ang r a)
  ((get 'make-from-mag-ang 'polar) r a))
```

Finally, this allows us to build a simple, data-directed system of operations:

```
(define (add x y) (apply-generic 'add x y))
(define (sub x y) (apply-generic 'sub x y))
(define (mul x y) (apply-generic 'mul x y))
(define (div x y) (apply-generic 'div x y))
```

Similar methods hold for the other types of data objects in our system.

Here is an example that traces through the use of generic operations under this method.

```
(define 2+5i (make-complex-from-real-imag
              (make-scheme-number 2)
              (make-scheme-number 5)))

(define 4+6i (make-complex-from-real-imag
              (make-scheme-number 4)
              (make-scheme-number 6)))

(define 1+3i (make-complex-from-real-imag
              (make-scheme-number 1)
              (make-scheme-number 3)))

(define 2+3i (make-complex-from-real-imag
              (make-scheme-number 2)
              (make-scheme-number 3)))


(define 2+5i/4+6i (make-rational 2+5i 4+6i))

(define 1+3i/2+3i (make-rational 1+3i 2+3i))
```

Now we can try manipulating these objects:

```
(add 2+5i/4+6i  1+3i/2+3i)

(apply-generic
   'add
   '(rational (complex
                (rect (scheme-number 2) (scheme-number 5)))
              (complex
                (rect (scheme-number 4) (scheme-number 6))))
   '(rational (complex
                (rect (scheme-number 1) (scheme-number 3)))
              (complex
                (rect (scheme-number 2) (scheme-number 3)))))
```

Remember that our numbers have types on them, so we do a get into the table based on `'add` and `(rational rational)`.

The resulting procedure is applied to the contents of the two numbers:

```
(attach-tag
  'rational
  (add-rat
   '((complex (rect (scheme-number 2) (scheme-number 5)))
     (complex (rect (scheme-number 4) (scheme-number 6))))
   '((complex (rect (scheme-number 1) (scheme-number 3)))
     (complex (rect (scheme-number 2) (scheme-number 3))))))

(attach-tag
  'rational
  (make-rat
    (add
     (mul
       '(complex (rect (scheme-number 2) (scheme-number 5)))
       '(complex (rect (scheme-number 2) (scheme-number 3))))
     (mul
       '(complex (rect (scheme-number 1) (scheme-number 3)))
       '(complex (rect (scheme-number 4) (scheme-number 6)))))
  (mul
    '(complex (rect (scheme-number 4) (scheme-number 6)))
    '(complex (rect (scheme-number 2) (scheme-number 3))))))
```

Now we can see the generic dispatching at work. The `muls` will use the table under `mul` and `(complex complex)` to get the right procedure, and apply to the objects, with the complex type stripped off. This eventually reduces to something like:

```
(attach-tag
   'rational
   (make-rat
     (add '(complex
             (polar (scheme-number 19.42) (scheme-number 124.4)))
           (complex
             (polar (scheme-number 14.14) (scheme-number 135.0))))
     '(complex
        (polar (scheme-number 36.76) (scheme-number 112.6)))))
```

and so on.

A major drawback to our system is that it cannot deal with expressions involving different types of objects. One way to fix this is to explicitly define procedures to deal with cross-types, for example:

```
;; to be included in the complex package
(define (add-complex-to-schemenum z x)
  (make-from-real-imag (+ (real-part z) x)
                       (imag-part z)))
(put 'add '(complex scheme-number)
     (lambda (z x)
        (tag (add-complex-to-schemenum z x))))
```

A better method would be to add coercion to our system, in which we allow for the possibility of forcing a data object of one type to convert to a data object of a different type:

```
(define (scheme-number->complex n)
  (make-complex-from-real-imag (contents n) 0))
```

We install these coercion procedures in a special coercion table, indexed under the names of the two types:

```
(put-coercion 'scheme-number
              'complex
              scheme-number->complex)
```

Then we simply change how we apply operators to arguments:

```
(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))
                 (arg1 (car args))
                 (arg2 (cadr args)))
             (let ((t1->t2 (get-coercion type1 type2))
                   (t2->t1 (get-coercion type2 type1)))
               (cond (t1->t2
                      (apply-generic op (t1->t2 arg1) arg2))
                     (t2->t1
                      (apply-generic op arg1 (t2->t1 arg2)))
                     (else
                      (error "No method" (list op type-tags))))))
           (error "No method for the given types"
                  (list op type-tags)))))))
```

When does coercion work well, and when does it fail?