Strategies for Managing Data Complexity

• Separate specification from implementation
• Procedural interface to data (enables alternative reps)
• Manifest typing (enables multiple representations)
• Generic operations
  - Dispatch on type
  - Table-driven interface (Data-Directed Programming)
• Packages
• Additivity
• Closure
• Coercion
• Message-Passing
Rectangular Package

;; Rectangular complex implementation...
(define (install-rectangular-package)
  ;; Internal rep: RepRect = Sch-Num X Sch-Num
  ;; internal procedures on RepRect...
  (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
  ;; External rep: Rectangular = 'rectangular X RepRect
  (define (tag x) (attach-tag 'rectangular x))
  ; Accessors
  (put 'real-part '(rectangular) real-part)
  (put 'imag-part '(rectangular) imag-part)
  (put 'magnitude '(rectangular) magnitude)
  (put 'angle '(rectangular) angle)
  ; Constructors: Sch-Num, Sch-Num -> Rectangular
  (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)
Constructor Interface Example

;; Generic constructors
(define (make-from-real-imag x y)
  ((get 'make-from-real-imag 'rectangular) x y))

Try this out for $3 + 4i$

;; Trace through substitution model...
(define my-z
  (make-from-real-imag 3 4))

(define my-z
  ((get 'make-from-real-imag 'rectangular) 3 4))

(define my-z
  ([lambda (x y) (tag (make-from-real-imag x y))] 3 4))

(define my-z
  (tag (make-from-real-imag 3 4)))

(define my-z (attach 'rectangular (cons 3 4)))
Generic Operation (Data-Directed) Example

(define (real-part z) (apply-generic 'real-part z))
(define (magnitude z) (apply-generic 'magnitude z))
...

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

An example generic accessor:

;; Trace through substitution model
(real-part my-z)

(apply-generic 'real-part my-z)

op -> 'real-part
args -> (my-z)
type-tags -> (rectangular) ;; why we installed
                           ;; as (rectangular) type!
proc -> [real-part proc from rectangular package]
==> 3
Complex Arithmetic (old)

;; Complex = Rectangular U Polar
;; add-complex: Complex, Complex -> Complex
(define (add-complex z1 z2)
  (make-from-real-imag
   (+ (real-part z1) (real-part z2))
   (+ (imag-part z1) (imag-part z2)))))

;; sub-complex: Complex, Complex -> Complex
(define (sub-complex z1 z2)
  (make-from-real-imag
   (- (real-part z1) (real-part z2))
   (- (imag-part z1) (imag-part z2)))))

;; mul-complex: Complex, Complex -> Complex
(define (mul-complex z1 z2)
  (make-from-mag-ang
   (* (magnitude z1) (magnitude z2))
   (+ (angle z1) (angle z2)))))

;; div-complex: Complex, Complex -> Complex
(define (div-complex z1 z2)
  (make-from-mag-ang
   (/ (magnitude z1) (magnitude z2))
   (- (angle z1) (angle z2))))
Complex Package - generic arithmetic

;;; the complex number package
(define (install-complex-package)
  ;; Internal Rep: RepComplex = Rectangular U Polar

  ;; import from rectangular and polar packages
  (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))

  ;; internal definitions...
  (define (add-complex z1 z2)
    (make-from-real-imag
      (+ (real-part z1) (real-part z2))
      (+ (imag-part z1) (imag-part z2))))
  (define (sub-complex z1 z2)
    (make-from-real-imag
      (- (real-part z1) (real-part z2))
      (- (imag-part z1) (imag-part z2))))
  (define (mul-complex z1 z2)
    (make-from-mag-ang
      (* (magnitude z1) (magnitude z2))
      (+ (angle z1) (angle z2))))
  (define (div-complex z1 z2)
    (make-from-mag-ang
      (/ (magnitude z1) (magnitude z2))
      (- (angle z1) (angle z2)))
Complex Package, cont’d

;;; interface to rest of system -- export to table
;;; External Rep: Complex = 'complex X RepComplex
(define (tag z) (attach-tag 'complex z))

(put 'add '(complex complex)
   (lambda (z1 z2) (tag (add-complex z1 z2)))))
(put 'sub '(complex complex)
   (lambda (z1 z2) (tag (sub-complex z1 z2)))))
(put 'mul '(complex complex)
   (lambda (z1 z2) (tag (mul-complex z1 z2)))))
(put 'div '(complex complex)
   (lambda (z1 z2) (tag (div-complex z1 z2)))))
; Constructors
(put 'make-from-real-imag 'complex
   (lambda (x y) (tag (make-from-real-imag x y)))))
(put 'make-from-mag-ang 'complex
   (lambda (r a) (tag (make-from-mag-ang r a)))))
'done)
Generic Arithmetic

(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))

Ordinary Number Package

;;; the ordinary number package
(define (install-number-package)
    ;; Internal rep: RepNum = Sch-Num
    ;; internal procedures -- just use Scheme!

    ;; External rep: Number = 'number X RepNum
    (define (tag x) (attach-tag 'number x))
    (put 'make 'number tag)
    (put 'add '(number number)
        (lambda (x y) (tag (+ x y))))
    (put 'sub '(number number)
        (lambda (x y) (tag (- x y))))
    (put 'mul '(number number)
        (lambda (x y) (tag (* x y))))
    (put 'div '(number number)
        (lambda (x y) (tag (/ x y))))
    'done)

;;; External constructor for ordinary numbers:
;;; Sch-Num --> Number
(define (create-number x)
    ((get 'make 'number) x))
Rational Number Package

;;; the rational number package
(define (install-rational-package)
  ;; Internal rep: RepRat = Sch-Num X Sch-Num
  ;; internal procedures on RepRat
  (define (make-rat n d) (cons n d))
  (define (numer x) (car x))
  (define (denom x) (cdr x))
  (define (add-rat x y)
    (make-rat (+ (* (numer x) (denom y))
                (* (denom x) (numer y)))
               (* (denom x) (denom y))))
  (define (sub-rat x y)
    (make-rat (- (* (numer x) (denom y))
                (* (denom x) (numer y)))
               (/ (denom x) (denom y))))
  (define (mul-rat x y)
    (make-rat (* (numer x) (numer y))
               (* (denom x) (denom y))))
  (define (div-rat x y)
    (make-rat (* (numer x) (denom y))
               (* (denom x) (numer y))))
Rational Package, cont’d

;;; External rep: Rational = 'rational X RepRat
(define (tag x) (attach-tag 'rational x))
(put 'make 'rational
  (lambda (n d) (tag (make-rat n d))))
(put 'add '(rational rational)
  (lambda (x y) (tag (add-rat x y))))
(put 'sub '(rational rational)
  (lambda (x y) (tag (sub-rat x y))))
(put 'mul '(rational rational)
  (lambda (x y) (tag (mul-rat x y))))
(put 'div '(rational rational)
  (lambda (x y) (tag (div-rat x y))))
'done)

;;; External constructor interface
(define (create-rational n d)
  ((get 'make 'rational) n d))
Problem: Rational with Complex

\[ \frac{3}{(4+5i)} + \frac{1}{(2+3i)} \]

Approach: Generic Number

;; Generic Numbers:

;; Generic-Num = Rational U Complex U Number

Allow the numerator and denominator in our rational package to be Generic-Nums, not just Sch-Nums!
Rational Package - "Genericized"

;;;; the rational number package
(define (install-rational-package)

;;;; Internal rep: RepRat =Generic-Num X Generic-Num
;;;; internal procedures on RepRat
(define (make-rat n d) (cons n d))
(define (numer x) (car x))
(define (denom x) (cdr x))
(define (add-rat x y)
    (make-rat (+ (mul (numer x) (denom y))
                 (mul (denom x) (numer y)))
        (mul (denom x) (denom y))))

(define (sub-rat x y)
    (make-rat (- (mul (numer x) (denom y))
                 (mul (denom x) (numer y)))
        (mul (denom x) (denom y))))

(define (mul-rat x y)
    (make-rat (* (numer x) (numer y))
        (mul (denom x) (denom y))))

(define (div-rat x y)
    (make-rat (* (numer x) (denom y))
        (mul (denom x) (numer y))))

;;;; Interface to rest of system
...
'done)
Rational/Complex Example

\[
\frac{3}{4 + 5i} + \frac{1}{2 + 3i}
\]

(define z1 (create-rational

    (create-number 3)

    (make-from-real-imag (create-number 4)
        (create-number 5)))

(define z2 (create-rational

    (create-number 1)

    (make-from-real-imag (create-number 2)
        (create-number 3)))

(mul z1 z2)

==> (rational (number 4)

    (complex (rectangular (number 6)
        (number 8))))

Problem: Mixed Types

An easy example:

\[3 \times \frac{1}{2}\]

No element in the table!

Approach: Mixed-Type Procedures

;;; In the rational package...
;;; mul-num-rat: RepNum, RepRat -> Rational
(define (mul-num-rat n r)
  (tag (mul-rat (make-rat n (create-number 1))
              r))
  (put 'mul '(number rational) mul-num-rat)

Coercion

(define (number->rational x)
  (create-rational x (create-number 1)))
Coercion Table

;;; (put-coercion <from-type> <to-type> <procedure>)
(put-coercion 'number 'rational number->rational)

(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 ((t1 (car type-tags))
              (t2 (cadr type-tags))
              (arg1 (car args))
              (arg2 (cadr args)))
            (let ((t1->t2 (get-coercion t1 t2))
              (t2->t1 (get-coercion t2 t1)))
              (cond (t1->t2
                (apply-generic op
                  (t1->t2 a1)
                  a2))
                (t2->t1
                  (apply-generic op
                    a1
                    (t2->t1 a2)))
                (else (error "No method"))))))
      (error "No method"))))
Polynomials

\[ 2x^5 + 7x + 3 \quad \text{to which we want to add } 3x \]

;; Term = <order> X <coefficient>
;; = Pos-Integer X Generic-Num
;;
;; make-term: Pos-Integer, GenericNum -> Term
;; order: Term -> Pos-Integer
;; coeff: Term -> Generic-Num

;; TermList = empty-termlist U (Term X Termlist)
;;
;; the-empty-termlist: () -> empty-termlist
;; empty-termlist?: TermList -> Bool
;; adjoin-term: Term, TermList -> TermList
;; first-term: TermList -> Term
;; rest-terms: TermList -> TermList
Polynomial Example

\(2x^5 + 7x + 3\) becomes

\[
\text{(define p1}
\left[
\text{(make-polynomial 'x}
\left[
\text{(adjoin-term (make-term 5 2) (adjoin-term (make-term 7 1) (adjoin-term (make-term 3 0) (the-empty-termlist)))))}
\right]
\right)
\]

\text{p1} \\
\rightarrow \ (\text{polynomial} \ x \ (5 \ 2) \ (7 \ 1) \ (3 \ 0))
Polynomial Package

(define (install-polynomial-package)
  ;; Internal Rep: RepPoly = ???
  ;; internal procedures
  (define (make-poly variable term-list) ...)
  (define (add-poly p1 p2) ...)
  (define (mul-poly p1 p2) ...)

  ;; representations used internally
  ;; for terms and term-lists
  ...

  ;; External Rep: Polynomial = 'polynomial X RepPoly
  (define (tag x) (attach-tag 'polynomial x))
  (put 'add '(polynomial polynomial)
       (lambda (p1 p2) (tag (add-poly p1 p2)))))
  (put 'mul '(polynomial polynomial)
       (lambda (p1 p2) (tag (mul-poly p1 p2)))))
  (put 'make 'polynomial
       (lambda (var terms) (tag (make-poly var terms)))))
  'done)