MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.001—Structure and Interpretation of Computer Programs
Fall Semester, 1998

Lecture Notes, September 22 – Data Abstraction

Pair Abstraction

1. Constructor

   ; cons: AnyType, AnyType → Pair
   `(cons <x> <y>) ==> <p>`

2. Accessors

   ; car, cdr: Pair → AnyType
   `(car <p>)`
   `(cdr <p>)`

3. Contract

   `(car (cons <x> <y>)) ==> <x>`
   `(cdr (cons <x> <y>)) ==> <y>`

4. Operations

   ; pair?: AnyType → Sch-Bool
   `(pair? <p>)`

5. **Abstraction Barrier** : Say nothing about implementation of pairs!

Rational Number Abstraction

1. Constructor

   ; make-rat: Int, Int → Rat
   `(make-rat <n> <d>) ==> <r>`

2. Accessors

   ; numer, denom: Rat → Int
   `(numer <r>)`
   `(denom <r>)`

3. Contract

   `(numer (make-rat <n> <d>)) ==> <n>`
   `(denom (make-rat <n> <d>)) ==> <d>`

4. Layered Operations

   `(print-rat <r>)`

5. **Abstraction Barrier:** Say nothing about implementation of Rat!

6. Concrete Representation & Implementation

   ```
   (define (make-rat n d) (cons n d))
   (define (numer r) (car r))
   (define (denom r) (cdr r))
   ```

Layered Rat Operations

```
; +rat: Rat, Rat → Rat
(define (+rat x y)
  (make-rat (+ (* (numer x) (denom y))
               (* (numer y) (denom x)))
            (* (denom x) (denom y))))

; *rat: Rat, Rat → Rat
(define (*rat x y)
  (make-rat (* (numer x) (numer y))
            (* (denom x) (denom y))))
```

"Rationalizing" Implementation

```
(define (numer r)
  (let ((g (gcd (car r) (cdr r))))
    (/ (car r) g)))

(define (denom r)
  (let ((g (gcd (car r) (cdr r))))
    (/ (cdr r) g)))

(define (make-rat n d)
  (cons n d))

(define (gcd a b)
  (if (= b 0)
    a
    (gcd b (remainder a b))))
```

Alternative "Rationalizing" Implementation

```
(define (numer r) (car r))

(define (denom r) (cdr r))

(define (make-rat n d)
  (let ((g (gcd n d)))
    (cons (/ n g)
          (/ d g))))
```

Cash Register

```
; 1. Constructor
(define (make-cash-register q d n p)
 (list (make-tray q (make-rat 1 4))
       (make-tray d (make-rat 1 10))
       (make-tray n (make-rat 1 20))
       (make-tray p (make-rat 1 100))))

; 2. Accessors
(define (register-quarters reg)
  (num-coins (car reg)))
(define (register-dimes reg)
  (num-coins (cadr reg)))

; 3. Operations
(define (num-coins-in-register reg)
  (accumulate + 0 (map num-coins reg)))

(define (cash-in-register reg)
  (accumulate +rat (make-rat 0 1)
              (map cash-in-tray reg)))
```

Coin Tray

```
; 1. Constructor
; make-tray: Int, Rat → Tray
(define (make-tray count denom)
   ; private implementation
   ; Tray = Int X Rat
  (cons count denom))

; 2. Accessors
; num-coins: Tray → Int
(define (num-coins tray) (car tray))

; demon-coins: Tray → Rat
(define (denom-coins tray) (cdr tray))

; Layered Operations
; cash-in-tray: Tray → Rat
(define (cash-in-tray tray)
  (*rat (make-rat (num-coins tray) 1)
        (denom-coins tray)))
```