Code Design, Debugging, and Types

Comp101

Deniz Yuret

Overview

- Writing beautiful code
- Types
- Code design
- Debugging

Writing Beautiful Code

The International Obfuscated C Code Contest

- Symmetry (Westley 1987)
- What does it do? (Westley 1988)
- She loves me not. (Westley 1990)
- Where am I (Westley 1992)

Better Documented Code

```
(define sqrt-helper
  (lambda (X guess)
    ;;; compute approximate square root by
    ;;; successive refinement, guess is current
    ;;; approximation, X is number whose square
    ;;; root we are seeking.
    ;;; Type: (number, number) -> number
    ;;; constraint: guess^2 == X
    (if (good-enuf? X guess)     ; can we stop?
        guess                    ; if yes, return
        (sqrt-helper X
                     (improve X guess)
                       ; if not, then get better guess
                       ; and repeat process
                     ))))
```

Types

- `15`
- number

Types

- `15`
- `"hi"`

- number
- string

Types

- `15`
- `"hi"`
- `square`

- number
- string
- number $\rightarrow$number

Types

- `15`
- `"hi"`
- `square`
- `>`

- number
- string
- number $\rightarrow$number
- number, number $\rightarrow$boolean

Types

- `15`
- `"hi"`
- `square`
- `>`
- `(cons 1 2)`

- number
- string
- number →number
- number, number →boolean
- Pair <number, number>

Types

- `15`
- `"hi"`
- `square`
- `>`
- `(cons 1 2)`
- `(list 1 2 "hi")`

- number
- string
- number $\rightarrow$number
- number, number $\rightarrow$boolean
- Pair <number, number>
- List <number or string>

Types

```
(lambda (a b c) (if (> a 0) (+ b c) (- b c)))
```

Types

```
(lambda (a b c) (if (> a 0) (+ b c) (- b c)))
```

number, number, number $\rightarrow$number

Types

```
(lambda (p) (if p "hi""bye"))
```

Types

```
(lambda (p) (if p "hi""bye"))
```

boolean →string

Types

```
(* 3.14 (+ 3 5))
```

Types

```
(* 3.14 (+ 3 5))
```

number

Types

```
(define fixed-point
  (lambda (f x)
    (if (close-enuf? x (f x))
        x
        (fixed-point f (f x)))))
```

Types

```
(define fixed-point
  (lambda (f x)
    (if (close-enuf? x (f x))
        x
        (fixed-point f (f x)))))
```

(number →number), number →number

Types

```
(define deriv
  (lambda (f)
    (lambda (x)
      (/ (- (f (+ x dx))
            (f x))
         dx))))

((deriv square) 5) ==> 10.0001
```

Types

```
(define deriv
  (lambda (f)
    (lambda (x)
      (/ (- (f (+ x dx))
            (f x))
         dx))))
```

```
((deriv square) 5) ==> 10.0001
```

(number $\rightarrow$number) $\rightarrow$(number $\rightarrow$number)

Types

```
(define unit-circle
  (lambda (t)
    (make-point (sin (* 2pi t))
                (cos (* 2pi t)))))
```

Types

```
(define unit-circle
  (lambda (t)
    (make-point (sin (* 2pi t))
                (cos (* 2pi t)))))
```

number[0,1] $\rightarrow$point

Types

What is the type of compose?

```
(define compose
  (lambda (f g)
    (lambda (x)
      (f (g x)))))
```

Types

Could it be (number →number), (number →number) →(number →number)?

```
(define compose
  (lambda (f g)
     (lambda (x)
        (f (g x)))))

((compose square double) 3)
;Value: 36

((compose double square) 3)
;Value: 18
```

Types

But we can use it for non-numeric procedures!

```
(define compose
  (lambda (f g)
     (lambda (x)
        (f (g x)))))

((compose
  (lambda (p) (if p "hi" "bye"))   ; boolean -> string
  (lambda (x) (> x 0)))            ; number -> boolean
 -5)   ; number

;Value: "bye"                      ; Result: string
```

Types

Can we use it for any two procedures?

```
(define compose
  (lambda (f g)
     (lambda (x)
        (f (g x)))))

((compose < square) 3)
;The procedure #[compound-procedure 6 <] has been called
;with 1 argument it requires exactly 2 arguments.

((compose square double) "hi")
;The object "hi", passed as the first argument to
;integer-add is not the correct type.
```

Types

The types need to match: use *type variables*.

```
(define compose
  (lambda (f g)
    (lambda (x)
      (f (g x)))))
```

$(B \rightarrow C), (A \rightarrow B) \quad \rightarrow \quad (A \rightarrow C)$

Code Design

A common pattern: repeatedly applying the same procedure.

```
(define mul
  (lambda (a b)
    (if (= b 0)
        0
        (+ a (mul a (- b 1))))))

(define exp
  (lambda (a b)
    (if (= b 0)
        1
        (mul a (exp a (- b 1))))))
```

Code Design

A common pattern: repeatedly applying the same procedure.

```
(define mul
  (lambda (a b)
    ((repeated
       (lambda (x) (+ x a))
       b)
     0)))

(define exp
  (lambda (a b)
    ((repeated
       (lambda (x) (mul x a))
       b)
     1)))
```

Code Design

A common pattern: repeatedly applying the same procedure.

```
(define mul
  (lambda (a b)
    ((repeated
       (lambda (x) (+ x a))
       b)
```

- What is the type of `repeated`?

Code Design

A common pattern: repeatedly applying the same procedure.

```
(define mul
  (lambda (a b)
    ((repeated
       (lambda (x) (+ x a))
       b)
```

- What is the type of `repeated`?
- (A $\rightarrow$A), Integer $\rightarrow$ (A $\rightarrow$A)

Code Design

repeated: (A →A), Integer → (A →A)

```
(define repeated
  (lambda (proc n)
    (if (= n 0)
        [ ??? ]
        [ ??? (repeated ??? (- n 1) ??? ]))
```

Code Design

repeated: (A →A), Integer → (A →A)

```
(define repeated
  (lambda (proc n)
    (if (= n 0)
        (lambda (x) x)
        [ ??? (repeated ??? (- n 1) ??? ]))
```

Code Design

repeated: (A →A), Integer → (A →A)

```
(define repeated
  (lambda (proc n)
    (if (= n 0)
        (lambda (x) x)
        (lambda (x)
          [ ??? (repeated ??? (- n 1) ??? ]))
```

Code Design

repeated: (A →A), Integer → (A →A)

```
(define repeated
  (lambda (proc n)
    (if (= n 0)
        (lambda (x) x)
        (lambda (x)
          (proc
            [ ??? (repeated ??? (- n 1) ??? ]))))
```

Code Design

repeated: (A →A), Integer → (A →A)

```
(define repeated
  (lambda (proc n)
    (if (= n 0)
        (lambda (x) x)
        (lambda (x)
          (proc
            ((repeated proc (- n 1)) x))))))
```

Debugging

```
(define foo
  (lambda (n)
    (if (= n 0)
        bar
        (+ n (foo (- n 1))))))

(foo 3)
```

Debugging

```
(define foo
  (lambda (n)
    (if (= n 0)
        bar
        (+ n (foo (- n 1))))))

(foo 3)

;Unbound variable: bar
;Type D to debug error, Q to quit back to REP loop:
```

Debugging

$$\sin x \approx x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots$$

```
;;; Compute sine function using mathematical approximation
(define (sine x)
  (define (aux x n current)
    (let ((next (/ (expt x n) (fact n))))
      ;; compute next term
      (if (small-enuf? next)  ;; if small
          current             ;; just return current guess
          (aux x (+ n 1) (+ current next))
            ;; otherwise, create new guess
          )))
```

Debugging: $\sin x \approx x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots$

```
;;; (2) Add some print statements
(define (sine x)
  (define (aux x n current)
    (display "n␣is␣") (display n)
    (display "␣current␣is␣") (display current) (newline)

    (let ((next (/ (expt x n) (fact n))))
      ;; compute next term
      (if (small-enuf? next)  ;; if small
          current             ;; just return current guess
          (aux x (+ n 1) (+ current next))
            ;; otherwise, create new guess
          )))
  (aux x 1 0))
```

Debugging: $\sin x \approx x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots$

```
;;; (3) Fix the increment
(define (sine x)
  (define (aux x n current)
    (display "n is ") (display n)
    (display " current is ") (display current) (newline)

    (let ((next (/ (expt x n) (fact n))))
      ;; compute next term
      (if (small-enuf? next)  ;; if small
          current             ;; just return current guess
          (aux x (+ n 2) (+ current next))
                ;;;;;;;
            ;; otherwise, create new guess
          )))
  (aux x 1 0))
```

Debugging: $\sin x \approx x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots$

```
;;; (4) Add alternating sign argument
(define (sine x)
  (define (aux x n current sign)
    (display "n␣is␣") (display n)
    (display "␣current␣is␣") (display current) (newline)

    (let ((next (/ (expt x n) (fact n)))) ;; next term
      (if (small-enuf? next)  ;; if small
          current             ;; just return current guess
          (aux x              ;; otherwise, create new guess
               (+ n 2)
               (+ current (* sign next))
               (* -1 sign)))))
  (aux x 1 0))
```

Debugging: $\sin x \approx x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots$

```
;;; (5) Add sign argument to initial call
(define (sine x)
  (define (aux x n current sign)
    (display "n is ") (display n)
    (display " current is ") (display current) (newline)

    (let ((next (/ (expt x n) (fact n)))) ;; next term
      (if (small-enuf? next)  ;; if small
          current             ;; just return current guess
          (aux x              ;; otherwise, create new guess
               (+ n 2)
               (+ current (* sign next))
               (* -1 sign))
          )))
  (aux x 1 0 -1))
```

Debugging: $\sin x \approx x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots$

```
;;; (6) Fix the initial sign argument
(define (sine x)
  (define (aux x n current sign)
    (display "n is ") (display n)
    (display " current is ") (display current) (newline)

    (let ((next (/ (expt x n) (fact n)))) ;; next term
      (if (small-enuf? next)  ;; if small
          current             ;; just return current guess
          (aux x              ;; otherwise, create new guess
               (+ n 2)
               (+ current (* sign next))
               (* -1 sign))
          )))
  (aux x 1 0 1))
```