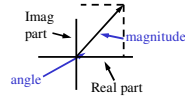


## Tagged Data

- Tag: a symbol in a data structure that identifies its type
- Why we need tags
- Extended example: evaluating arithmetic expressions

238

## Manipulating complex numbers



- Complex number has:
- real and imaginary part (Cartesian)
  - magnitude and angle (polar)

Addition is easier in  
Cartesian coordinates

```
(define (+c z1 z2)
  (make-complex-from-rect (+ (real z1) (real z2))
    (+ (imag z1) (imag z2))))
```

Multiplication is easier in  
polar coordinates

```
(define (*c z1 z2)
  (make-complex-from-polar (* (mag z1) (mag z2))
    (+ (angle z1) (angle z2))))
```

338

## Bert's data structure

```
(define (make-complex-from-rect rl im) (list rl im))
(define (make-complex-from-polar mg an)
  (list (* mg (cos an))
    (* mg (sin an))))
```

Note conversion to  
rectangular form  
before storing

```
(define (real cx) (first cx))
(define (imag cx) (second cx))
(define (mag cx) (sqrt (+ (square (real cx))
  (square (imag cx)))))
(define (angle cx) (atan (imag cx) (real cx)))
```

Need to do  
some  
computation  
since stored in  
rectangular  
form

438

## Ernie's data structure

```
(define (make-complex-from-rect rl im)
  (list (sqrt (+ (square rl) (square im)))
    (atan im rl)))
(define (make-complex-from-polar mg an) (list mg an))
```

Note conversion to  
polar form before  
storing

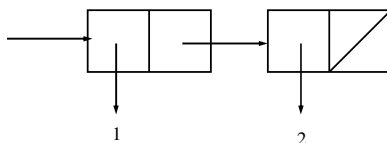
```
(define (real cx) (* (mag cx) (cos (angle cx))))
(define (imag cx) (* (mag cx) (sin (angle cx))))
(define (mag cx) (car cx))
(define (angle cx) (cadr cx))
```

Need to do  
some  
computation  
since stored in  
polar form

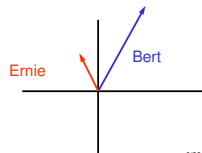
538

## Whose number is it?

- Suppose we pick up the following object



- What number does this represent?



638

## Labeled complex numbers

```
(define (make-complex-from-rect rl im)
  (list 'rect rl im))
(define (make-complex-from-polar mg an)
  (list 'polar mg an))
```

```
(define (tag obj) (first obj))
```

```
(define (real z)
  (cond ((eq? (tag z) 'rect) (second z))
    ((eq? (tag z) 'polar) (* (second z) ;mag
      (cos (third z)))) ;angle
    (else (error "unknown form of object"))))
```

738

## The concept of a tag

- Tagged data =
  - attach an identifying symbol to all nontrivial data values that indicates the type of the value
  - always check the tag before operating on the data

```
(define (make-point x y)
  (list 'point x y))

(define (make-complex-from-real r1 im)
  (list 'real r1 im))

(define (make-key modulus exponent)
  (list 'RSA-key modulus exponent))
```

8/38

## Benefits of tagged data

- **data-directed programming:** functions that decide what to do based on argument types

• example: in a graphics program  
**area: triangle|square|circle -> number**

- **defensive programming:** functions that fail gracefully if given bad arguments

– **much better** to give an error message than to return garbage!

9/38

## Example: Arithmetic evaluation

Create arithmetic expressions

```
(define an-expr (make-sum (make-sum 3 15) 20))
an-expr      ==> (+ (+ 3 15) 20)
(eval an-expr) ==> 38
```

Evaluate arithmetic expressions to reduce them to simpler form

Expressions might include values other than simple numbers

Ranges:

some unknown number between **min** and **max**  
arithmetic:  $[3,7] + [1,3] = [4,10]$

Limited precision values:

some value  $\pm$  some error amount  
arithmetic:  $(100 \pm 1) + (3 \pm 0.5) = (103 \pm 1.5)$

10/38

## Approach: start simple, then extend

- Characteristic of all software engineering projects
- Start with eval for numbers, then add support for ranges and limited-precision values
- Goal: build eval in a way that it will extend easily & safely
  - Easily: requires data-directed programming
  - Safely: requires defensive programming
- Process: multiple versions of eval
  - eval-1 Simple arithmetic, no tags
  - eval-2 Extend the evaluator, observe bugs
  - eval-3 through -7 Do it again with tagged data

11/38

## 1. Data abstraction for sums of numbers

```
(define (make-sum addend augend)
  ; type: Exp, Exp -> SumExp
  (list '+ addend augend))

(define (sum-exp? e)
  ; type: anytype -> boolean
  (and (pair? e) (eq? (car e) '+)))

(define (sum-addend sum) (cadr sum))
(define (sum-augend sum) (caddr sum))
; type: SumExp -> Exp
```

- the type Exp will be different in different versions of eval

12/38

## 1. Eval for sums of numbers

```
; Exp = number | SumExp
(define (eval-1 exp)
  ; type: Exp -> number
  (cond
    ((number? exp) exp) ; base case
    ((sum-exp? exp) ; recursive case
     (+ (eval-1 (sum-addend exp))
        (eval-1 (sum-augend exp))))
    (else
     (error "unknown expression " exp))))

(eval-1 (make-sum 4 (make-sum 3 5))) ==> 12
```

13/38

### Example in gory detail

`(eval-1 (make-sum 4 (make-sum 3 5))) ==> 12`

Sum-exp? checks this using eq?  
 Number? checks this  
 Sum-exp? checks this using eq?

```

(+ (eval-1 ...) (eval-1 ...))
(+ 4 (+ (eval-1 ...) (eval-1 ...)))
(+ 4 (+ 3 5))
  
```

14/98

### 2. Extend the abstraction to ranges (without tags)

```

; type: number, number -> range2
(define (make-range-2 min max) (list min max))

; type: range2 -> number
(define (range-min-2 range) (car range))
(define (range-max-2 range) (cadr range))

; type: range2, range2 -> range2
(define (range-add-2 r1 r2)
  (make-range-2
    (+ (range-min-2 r1) (range-min-2 r2))
    (+ (range-max-2 r1) (range-max-2 r2))))
  
```

15/98

### Detailed example of adding ranges

`(range-add-2 (make-range 3 7) (make-range 1 3))`

```

(make-range-2 (+ (make-range-2 4 10) (+ ...)))
  
```

This is a range

16/98

### 2. Eval for sums of numbers and ranges (broken!)

```

; Exp = number | range2 | SumExp
(define (eval-2 exp)
  ; type: Exp -> number|range2
  (cond
    ((number? exp) exp)
    ((sum-exp? exp)
     (let ((v1 (eval-2 (sum-addend exp)))
           (v2 (eval-2 (sum-augend exp))))
       (if (and (number? v1) (number? v2))
           (+ v1 v2) ; add numbers
           (range-add-2 v1 v2))) ; add ranges
    ((pair? exp) exp) ; a range
    (else (error "unknown expression " exp))))
  
```

17/98

### Why is eval-2 broken?

- Missing a case: sum of number and a range

`(eval-2 (make-sum 4 (make-range-2 4 6)))`  
`==> error: the object 4 is not a pair`

18/98

### 2. Eval for sums of numbers and ranges (broken!)

```

; Exp = number|range2|SumExp
(define (eval-2 exp) ; type: Exp -> number|range2
  (cond
    ((number? exp) exp)
    ((sum-exp? exp)
     (let ((v1 (eval-2 (sum-addend exp)))
           (v2 (eval-2 (sum-augend exp))))
       (if (and (number? v1) (number? v2))
           (+ v1 v2) ; add numbers
           (range-add-2 v1 v2))) ; add ranges
    ((pair? exp) exp) ; a range
    (else (error "unknown expression " exp))))
  
```

Range-add-2 expects two ranges, i.e. two lists!

19/98

### Why is eval-2 broken?

- Missing a case: sum of number and a range  

```
(eval-2 (make-sum 4 (make-range-2 4 6)))  
=> error: the object 4 is not a pair
```
- Not defensive: what if we add limited-precision values but forget to change eval-2 ?  

```
(define (make-limited-precision-2 val err)  
  (list val err))  
  
(eval-2 (make-sum  
  (make-range-2 4 6)  
  (make-limited-precision-2 10 1)))  
=> (14 7) correct answer: (13 17) or (15 2)
```

Key point – doesn't return an error, but gives us what appears to be a legitimate answer!

20/38

### Lessons from eval-2

- Common bug: calling a function on the wrong type of data
  - typos
  - brainos
  - changing one part of the program and not another
- Common result: the function returns garbage
  - Why? Primitive predicates like `number?` and `pair?` are ambiguous
  - Something fails later, but cause is hard to track down
  - Worst case: **program produces incorrect output!!**
- Next: how to use tagged data to ensure that the program halts immediately

21/38

### 3. Start again using tagged data

- Take another look at `SumExp` ... it's already tagged!  

```
(define sum-tag '+)  
; Type: Exp, Exp -> SumExp  
(define (make-sum addend augend)  
  (list sum-tag addend augend))  
; Type: anytype -> boolean  
(define (sum-exp? e)  
  (and (pair? e) (eq? (car e) sum-tag)))
```
- `sum-exp?` is not ambiguous: only true for things made by `make-sum` (assuming the tag `+` isn't used anywhere else)

22/38

### Data abstraction for numbers using tags

```
(define constant-tag 'const)  
  
; type: number -> ConstantExp  
(define (make-constant val)  
  (list constant-tag val))  
  
; type: anytype -> boolean  
(define (constant-exp? e)  
  (and (pair? e) (eq? (car e) constant-tag)))  
  
; type: ConstantExp -> number  
(define (constant-val const) (cadr const))
```

23/38

### 3. Eval for numbers with tags (incomplete)

```
; Exp = ConstantExp | SumExp  
(define (eval-3 exp) ; type: Exp -> number  
  (cond  
    ((constant-exp? exp) (constant-val exp))  
    ((sum-exp? exp)  
     (+ (eval-3 (sum-addend exp))  
        (eval-3 (sum-augend exp))))  
    (else (error "unknown expr type: " exp))))  
  
(eval-3 (make-sum (make-constant 3)  
  (make-constant 5))) ==> 8
```

• Not all nontrivial values used in this code are tagged

24/38

### 4. Eval for numbers with tags

```
; type: Exp -> ConstantExp  
(define (eval-4 exp)  
  (cond  
    ((constant-exp? exp) exp)  
    ((sum-exp? exp)  
     (make-constant  
      (+ (constant-val (eval-4 (sum-addend exp)))  
         (constant-val (eval-4 (sum-augend exp))))))  
    (else (error "unknown expr type: " exp))))  
  
(eval-4 (make-sum (make-constant 3)  
  (make-constant 5)))  
=> (constant 8)
```

There is that pattern of using selectors to get parts, doing something, then using constructor to reassemble

25/38

Make `add` an operation in the Constant abstraction

```
;type: ConstantExp, ConstantExp -> ConstantExp
(define (constant-add c1 c2)
  (make-constant (+ (constant-val c1)
                   (constant-val c2))))

; type: ConstantExp | SumExp -> ConstantExp
(define (eval-4 exp)
  (cond
   ((constant-exp? exp) exp)
   ((sum-exp? exp)
    (constant-add (eval-4 (sum-addend exp))
                  (eval-4 (sum-augend exp))))
   (else (error "unknown expr type: " exp))))
```

26/38

## Lessons from `eval-3` and `eval-4`

- standard pattern for a data abstraction with tagged data
  - a variable stores the tag
  - attach the tag in the constructor
  - write a predicate that checks the tag
    - determines whether an object belongs to the type of the abstraction
  - operations strip the tags, operate, attach the tag again
- must use tagged data everywhere to get full benefits
  - including return values

27/38

## 5. Same pattern: ranges with tags

```
(define range-tag 'range) [3, 7]
; type: number, number -> RangeExp
(define (make-range min max)
  (list range-tag min max))

; type: anytype -> boolean
(define (range-exp? e)
  (and (pair? e) (eq? (car e) range-tag)))

; type: RangeExp -> number
(define (range-min range) (cadr range))
(define (range-max range) (caddr range))
```

28/38

## 5. Eval for numbers and ranges with tags

```
; Exp = ConstantExp | RangeExp | SumExp
(define (eval-5 exp) ; type: Exp -> ConstantExp|RangeExp
  (cond
   ((constant-exp? exp) exp)
   ((range-exp? exp) exp)
   ((sum-exp? exp)
    (let ((v1 (eval-5 (sum-addend exp)))
          (v2 (eval-5 (sum-augend exp))))
      (if (and (constant-exp? v1) (constant-exp? v2))
          (constant-add v1 v2)
          (range-add (val2range v1) (val2range v2))))))
   (else (error "unknown expr type: " exp))))
```

29/38

## Simplify `eval` with a data-directed add function

```
; ValueExp = ConstantExp | RangeExp
(define (value-exp? v)
  (or (constant-exp? v) (range-exp? v)))

; type: ValueExp, ValueExp -> ValueExp
(define (value-add-6 v1 v2)
  (if (and (constant-exp? v1) (constant-exp? v2))
      (constant-add v1 v2)
      (range-add (val2range v1) (val2range v2))))

; val2range: if argument is a range, return it
; else make the range [x x] from a constant x
; This is called coercion
```

30/38

## Use type coercion to turn constants into ranges

```
(define (val2range val)
  ; type: ValueExp -> RangeExp
  (if (range-exp? val)
      val ; just return range
      (make-range (constant-val val)
                  (constant-val val))))
```

31/38

## 6. Simplified eval for numbers and ranges

```
; ValueExp = ConstantExp | RangeExp
; Exp = ValueExp | SumExp
(define (eval-6 exp)
  ; type: Exp -> ValueExp
  (cond
    ((value-exp? exp) exp)
    ((sum-exp? exp)
     (value-add-6 (eval-6 (sum-addend exp))
                  (eval-6 (sum-augend exp))))
    (else (error "unknown expr type: " exp))))
```

32/38

## Compare eval-6 with eval-1

```
(define (eval-6 exp)
  (cond
    ((value-exp? exp) exp)
    ((sum-exp? exp)
     (value-add-6 (eval-6 (sum-addend exp))
                  (eval-6 (sum-augend exp))))
    (else (error "unknown expr type: " exp))))
```

- Compare to eval-1. It is just as simple!

```
(define (eval-1 exp)
  (cond
    ((number? exp) exp)
    ((sum-exp? exp)
     (+ (eval-1 (sum-addend exp))
        (eval-1 (sum-augend exp))))
    (else
     (error "unknown expression " exp))))
```

- This shows the power of **data-directed programming**

33/38

## Eval-7: adding limited-precision numbers

```
(define limited-tag 'limited)
(define (make-limited-precision val err)
  (list limited-tag val err))

; Exp = ValueExp | Limited | SumExp
(define (eval-7 exp)
  ; type: Exp -> ValueExp | Limited
  (cond
    ((value-exp? exp) exp)
    ((limited-exp? exp) exp)
    ((sum-exp? exp)
     (value-add-6 (eval-7 (sum-addend exp))
                  (eval-7 (sum-augend exp))))
    (else (error "unknown expr type: " exp))))
```

34/38

## Oops: value-add-6 is not defensive

```
(eval-7 (make-sum
         (make-range 4 6)
         (make-limited-precision 10 1)))
==> (range 14 16) WRONG
```

```
(define (value-add-6 v1 v2)
  (if (and (constant-exp? v1) (constant-exp? v2))
      (constant-add v1 v2)
      (range-add (val2range v1) (val2range v2))))
```

- Correct answer should have been (range 13 17) or (limited 15 2)

35/38

## What went wrong in value-add-6?

- limited-exp is not a constant, so falls into the alternative
- (limited 10 1) passed to val2range
- (limited 10 1) passed to constant-val, returns 10
- range-add called on (range 4 6) and (range 10 10)

```
(define (value-add-6 v1 v2)
  (if (and (constant-exp? v1) (constant-exp? v2))
      (constant-add v1 v2)
      (range-add (val2range v1) (val2range v2))))

(define (val2range val)
  (if (range-exp? val)
      val
      ; just return range
      (make-range (constant-val val) ; assumes constant
                  (constant-val val))))
```

36/38

## 7. Defensive version: check tags before operating

```
; type: ValueExp, ValueExp -> ValueExp
(define (value-add-7 v1 v2)
  (cond
    ((and (constant-exp? v1) (constant-exp? v2))
     (constant-add v1 v2))
    ((and (value-exp? v1) (value-exp? v2))
     (range-add (val2range v1) (val2range v2)))
    (else
     (error "unknown exp: " v1 " or " v2))))
```

- Rule of thumb:  
when checking types, use the else branch only for errors

37/38

### Lessons from `eval-5` through `eval-7`

- Data directed programming can simplify higher level code
- Using tagged data is only defensive programming if you check the tags
  - don't put code in the else branch of `if` or `cond`; make it signal an error instead
- Traditionally, operations and accessors don't check tags
  - They assume tags have been checked at the higher level
  - A check in `constant-val` would have trapped this bug
  - Be paranoid: add checks in your operations and accessors
    - The cost of redundant checks is usually trivial compared to the cost of your debugging time
  - Andy Grove: "only the paranoid survive"

38/38