Data Abstraction

Gul Agha
CS 421

Data should be specified via interfaces
- **Interface**
  - What the data represents
- **Implementation**
  - Specific representation of the data and code for operations using the data

Abstract Data Types: the data type represented by the interface.
The rest of program manipulates data only through operations specified by the interface.

Example: Natural numbers

\[
\lceil v \rceil \quad \text{the representation of } v
\]

zero = \lceil 0 \rceil

(iszero? \lceil v \rceil) = \#t \quad n = 0
\#f \quad n \neq 0

(succ \lceil n \rceil) = \lceil n + 1 \rceil \quad n \geq 0

(pred \lceil n \rceil) = \lceil n - 1 \rceil \quad n \geq 0

Representation Independence

- Client program should *not* rely on the representation of data or operations of an abstract data type.

(define plus
  (lambda (x y)
    (if (iszero? x)
        y
        (succ (plus (pred x) y)))))

Unary Representation (Church’s encoding)

\[
[0] = ()
\[n + 1\] = (cons \#t [n])
\]

(define zero ‘())
(define iszero? null?)
(define succ (lambda (n) (cons \#t n)))
(define pred cdr)
Scheme’s Number Representation

(define zero 0)
(define iszero? zero?)
(define succ (lambda (n) (+ n 1)))
(define (pred (lambda (n) (- n 1)))

Big Num Representation

Represent the numbers in base N:
\[ \lceil n \rceil = ( ) \text{ if } n = 0 \]
\[ \lceil n \rceil = (\text{cons } r \lceil q \rceil) \]
\[ n = qN + r \text{ where } 0 \leq r < N \]

Example: N = 16
[33] = (1 2)
[258] = (2 0 1)

Abstraction to Represent Inductive Data Types

- **Constructors**
  - Build elements of inductive data type
- **Predicate**
  - To test if value is of
    - Inductively defined type
  - Subtypes used in definition
- **Extractor**
  - To deconstruct the data type value

Example Data Type

\[ \langle \text{bintree} \rangle \rightarrow \]
\[ \langle \text{number} \rangle \]
\[ \langle \text{symbol} \rangle \langle \text{bintree} \rangle \langle \text{bintree} \rangle \]

bintree is either
- a number
- the cross-product of a symbol and two binary trees

Defining Data Types in Scheme

(define-datatype bintree bintree? (leaf-node (datum number?))
    (interior-node (key symbol?) (left bintree?) (right bintree?)))

Interface of the Data Type

- a 1-argument procedure which constructs a leaf-node
  - leaf-node tests its arguments with number?
- a 3-argument procedure which constructs an interior-node
  - Tests its first argument with symbol?
  - Tests its second and third argument with bintree?
Syntax

(define-datatype type-name
  predicate-name
  { (variant-name {(field-name predicate)}*)
    }*)

Using the Bintree Datatype

(define-datatype bintree
  bintree?
  (leaf-node
    (datum number?))
  (interior-node
    (key symbol?)
    (left bintree?)
    (right bintree?)
  )

(define leaf-sum
  (lambda (tree)
    (cases bintree tree
      (leaf-node ( n )  n)
      (interior-node
        (key left right)
        (+ (leaf-sum left)
           (leaf-sum right)))
    )))

Concrete Syntax

- Particular representation of an inductive data type
  - `<expression>`
  - `<identifier>`
  - `(lambda (<identifier>) <expression>)`
  - `( <expression> <expression>)`

Abstract Syntax

- Represents the actual structure
- Omits elements that are not necessary
- Represented as an ordered tree
  - Root is grammatical form (nonterminal)
  - Leaves are terminal
  - Tree represents the application of a rule

Example

```
(lambda (x) (f (f x)))
```

```
<expression>
  → <identifier>  var-exp (id)
  → (lambda (<identifier>) <expression>)
  → ( <expression> <expression>)
  → app-exp (rator rand)
```
Parsing Expressions

(define parse-expression
  (lambda (datum)
    (cond
      ((symbol? datum) (var-exp datum))
      ((pair? datum)
        (if (eqv? (car datum) 'lambda)
          (lambda-exp (caadr datum)
                      (parse-expression (caddr datum)))
          (app-exp
           (parse-expression (car datum))
           (parse-expression (cadr datum))))))))

A Richer Language

<expression>
  → <number>      lit-exp (datum)
  → <var-exp>    var-exp (id)
  → (if <expression> <expression> <expression>)
  → (lambda <identifier> <expression>)
  → (<expression> <expression>)
  → (app-exp <rator> <rand>)

Note: Some more useful higher order functions

- Reduce
- Zip

Reference

Also has fun web programming package in Scheme (go up in webpage)

Reduce

Successive reductions by a binary operator

Example

- (reduce-left - '(1 2 3 4 5))
  -13
- (reduce-right - '(1 2 3 4 5))
  3
- (reduce-left append
    (list (list 1 2 3) (list 'a 'b 'c)))
  (1 2 3 a b c)

reduce-right defined

(define reduce-right
  (lambda (f lst)
    (if (null? (cdr lst))
      (car lst)
      (f (car lst)
       (reduce-right f (cdr lst))))))
Trace of call to reduce right

> (reduce-right '(1 2 3 4 5))
| (reduce-right #<primitive:-> (1 2 3 4 5))
| | (reduce-right #<primitive:-> (2 3 4 5))
| | | (reduce-right #<primitive:-> (3 4 5))
| | | | (reduce-right #<primitive:-> (4 5))
| | | | | (reduce-right #<primitive:-> (5))
| | | | | | 15
| | | | | | | -1
| | | | | | | 4
| | | | | | | -2
| | | | | | | 3

Iterative Procedure for reduce-left

(define reduce-left
(lambda (f lst)
(reduce-help-left f (cdr lst) (car lst)))))

(define reduce-help-left
(lambda (f lst res)
(if (null? lst)
res
(reduce-help-left f (cdr lst)
(f res (car lst))))))

Sample execution of reduce-left

> (reduce-left - '(1 2 3 4 5))
| (reduce-help-left #<primitive:-> (2 3 4 5) 1)
| | (reduce-help-left #<primitive:-> (3 4 5) -1)
| | | (reduce-help-left #<primitive:-> (4 5) -4)
| | | | (reduce-help-left #<primitive:-> (5) -8)
| | | | | (reduce-help-left #<primitive:-> () -13)
| | | | | | -13
| | | | | |

Zip

Composes two lists of equal length to a single list by means of zipping their elements pair-wise

\[ (e_1, f_1, e_2, f_2, e_3, f_3, \ldots, e_n, f_n) \]

Example of zip

> (zip cons '(1 2 3) '(a b c))
| ((1 . a) (2 . b) (3 . c))

> (zip + '(1 2 3) '(4 5 6))
| (5 7 9)

Zip implementation

(define zip
(lambda (z lst1 lst2)
(if (null? lst1)
'(
(cons (z (car lst1) (car lst2))
(zip z (cdr lst1)
(cdr lst2))))))
So now FOLD. This is actually the one we've always hated most, because, apart from a few examples involving + or *, almost every time we see a FOLD call with a non-trivial function argument, we have to grab pen and paper and imagine the *result* of a function flowing back in as the *argument* to a function. Plus, there are *more* arguments coming in on the side! This is all absurdly complicated. Because almost all the examples of FOLD we found in practice could be written as a simple loop with an accumulator, this style should be preferred, perhaps with us providing a simple helper function to abstract away the boilerplate code. At any rate, FOLD must fold.

--The PLT Scheme Team