Data Types Representations

Gul Agha
CS 421
Representing Data Types

The same data type can have many representations.
We will use a running example of data type: *environments*
We will study three representations
- Procedural representation
- Abstract Syntax Tree representation
- Alternate Data Structure representation
Environments

- An environment associates a value with each element of a finite set of symbols.
Need to represent \{(s_1, s_2), \ldots, (s_n, s_n)\} 
where \(s_i\) are distinct symbols and \(v_i\) are any values.

**Interface:**

- \((\text{empty-env}) = [\emptyset]\)
- \((\text{apply-env} [f] s) = f(s)\)
- \((\text{extend-env}'(s_1 \ldots s_k)'(v_1 \ldots v_k) [f]) = [g]\)
  where \(g(s') = v_i\) if \(s' = s_i\) for some \(i, 1 \leq i \leq k\),
  \(= f(s')\) otherwise.
Procedural Representation

- Represent an environment as a procedure that takes a symbol and returns its associated value.
- Need to define procedures for:
  - *Constructing an* empty environment
  - *Extending* an environment
  - *Applying* an environment
Constructing an Empty Environment

(define empty-env
  (lambda ()
    (lambda (sym)
      (eopl:error 'apply-env
        "No binding for ~s" sym))))
Applying an environment

(define apply-env
  (lambda (env sym)
    (env sym)))
Extending an environment

(define extend-env
  (lambda (syms vals env)
    (lambda (sym)
      (let ((pos
        (list-find-position sym syms)))
        (if (number? pos)
          (list-ref vals pos)
          (apply-env env sym)))))))

Need to define auxiliary procedures to find corresponding positions and values.
Auxiliary Procedures I

(define list-find-position
  (lambda (sym los)
    (list-index
      (lambda (sym1)
        (eqv? sym1 sym)
      los))))
(define list-index
  (lambda (pred ls)
    (cond
      ((null? ls) #f)
      ((pred (car ls)) 0)
      (else
       (let ((list-index-r (list-index pred (cdr ls))))
        (if (number? list-index-r)
            (+ list-index-r 1)
            #f)))))))
General Idea for building procedural representations

- Identify the lambda expressions in the client code whose evaluation yields values of the data type
  - E.g., where are environments constructed
- Create a constructor procedure for each type of use
- Parameters of the constructor are the free variables in the use
  - E.g. if you are creating a stack, there may be checks on the max size
- Replace each lambda expression in the client code by an invocation of the corresponding constructor
- Fix invocations...
Abstract Syntax Representation

Describe the data structure by a grammar
Example: observe environments are built by
- Creating an empty environment
- Extending the empty environment
Inductive definition of Environment

<env-rep>
  \to (empty-env)
  empty-env-record
  \to (extended-env ({<symbol>}*)
      ({{<value>}}*)
      <env-rep>)

extended-env-record
(syms vals env)
Environments as Variant Record

(define-datatype environment environment?  
(empty-env-record)  
(extended-env-record  
 (syms (list-of symbol?))  
 (vals (list-of scheme-value?))  
 (env environment?)))

(define scheme-value? (lambda (v) #t))
The Interface Procedures: Constructing the environment

(define empty-env
  (lambda ()
    (empty-env-record)))

(define extend-env
  (lambda (syms vals env)
    (extended-env-record
      syms vals env)))
(define apply-env
  (lambda (env sym)
    (cases environment env
      (empty-env-record ()
        (eopl:error 'apply-env ... ))
      (extended-env-record (syms vals env)
        (let ((pos (list-find-position sym syms)))
          (if (number? pos)
              (list-ref vals pos)
              (apply-env env sym)))))))
Alternate Representation

- Can optimize on the tree structure
- Example:
  - For environments, can have lists of symbols and vectors of values

<env-rep>
  
  \( \rightarrow ( ) \)
  
  \( \rightarrow ( ( (\{<\text{symbol}>\}\} *) (\{<\text{value}>\} *) ) \).
  
<env-rep>
A Queue Abstraction

- (create-queue)
- (queue-get-reset-operation q)
- (queue-get-empty?-operation q)
- (queue-get-enqueue-operation q)
- (queue-get-dequeue-operation q)