Data abstraction, revisited

- Design tradeoffs:
  - Speed vs robustness
  - Modularity
  - Ease of maintenance
- Table abstract data type: 3 versions
- No implementation of an ADT is necessarily "best"
- Abstract data types hide information, in types as well as in the code

Table: a set of bindings

- Binding: a pairing of a key and a value
- Abstract interface to a table:
  - Make: create a new table
  - Put! key value: insert a new binding
  - Get key: look up the key, return the corresponding value
- This definition IS the table abstract data type
- Code shown later is a particular implementation of the ADT

Examples of using tables

- Values associated with keys might be data structures
- Values might be shared by multiple structures

Traditional LISP structure: association list

- A list where each element is a list of the key and value.
- Represent the table

<table>
<thead>
<tr>
<th>Age</th>
<th>Job</th>
<th>Pay</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1999</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1998</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

as the alist: `((x 15) (y 20))`

Alist operation: find-assoc

```
(define (find-assoc key alist)
  (cond
    ((null? alist) #f)
    ((equal? key (caar alist)) (cadar alist))
    (else (find-assoc key (cdr alist))))
```

```
(define a1 '((x 15) (y 20)))
(find-assoc 'y a1) ==> 20
```

An aside on testing equality

- = tests equality of numbers
- Eq? Tests equality of symbols
- Equal? Tests equality of symbols, numbers or lists of symbols and/or numbers that print the same
Alist operation: add-assoc

\[
\text{add-assoc} \quad \text{cons} \quad \text{list} \quad \text{key} \quad \text{val} \quad \text{alist}
\]

\[
\text{define (add-assoc key val alist)}
\]

\[
\text{(cons (list key val) alist))}
\]

\[
\text{define a2 (add-assoc 'y 10 a1)}
\]

\[
a2 \implies ((y 10) (x 15) (y 20))
\]

\[
\text{(find-assoc 'y a2) \implies 10}
\]

We say that the new binding for y “shadows” the previous one.

Alists are not an abstract data type

\[
\text{• Missing a constructor:}
\]

\[
\text{• Used quote or list to construct}
\]

\[
\text{(define a1 '((x 15) (y 20)))}
\]

\[
\text{• There is no abstraction barrier: the implementation is exposed.}
\]

\[
\text{• User may operate on alists using standard list operations.}
\]

\[
\text{(filter (lambda (a) (< (cadr a) 16)) a1))}
\]

\[
\implies ((x 15))
\]

Why do we care that Alists are not an ADT?

\[
\text{• Modularity is essential for software engineering}
\]

\[
\text{• Build a program by sticking modules together}
\]

\[
\text{• Can change one module without affecting the rest}
\]

\[
\text{• Alists have poor modularity}
\]

\[
\text{• Programs may use list ops like filter and map on alists}
\]

\[
\text{• These ops will fail if the implementation of alists change}
\]

\[
\text{• Must change whole program if you want a different table}
\]

\[
\text{• To achieve modularity, hide information}
\]

\[
\text{• Hide the fact that the table is implemented as a list}
\]

\[
\text{• Do not allow rest of program to use list operations}
\]

\[
\text{• ADT techniques exist in order to do this}
\]

Table 1: Table ADT (implemented as an Alist)

\[
\text{define table1-tag 'table1}
\]

\[
\text{(define (make-table1) (cons table1-tag nil))}
\]

\[
\text{(define (table1-get tbl key) (find-assoc key (cdr tbl)))}
\]

\[
\text{(define (table1-put! tbl key val) (set-cdr! tbl (add-assoc key val (cdr tbl))))}
\]

Compound Data

\[
\text{• constructor:}
\]

\[
\text{(cons x y) creates a new pair p}
\]

\[
\text{• selectors:}
\]

\[
\text{(car p) returns car part of pair}
\]

\[
\text{(cdr p) returns cdr part of pair}
\]

\[
\text{• mutators:}
\]

\[
\text{(set-car! p new-x) changes car pointer in pair}
\]

\[
\text{(set-cdr! p new-y) changes cdr pointer in pair}
\]

\[
\text{Pair, anytype -> undef -- side-effect only!}
\]

Example 1: Pair/List Mutation

\[
\text{(define a (list 1 2))}
\]

\[
\text{(define b a)}
\]

\[
a \implies (1 2)
\]

\[
b \implies (1 2)
\]

\[
\text{(set-car! a 10)}
\]

\[
\text{b \implies (10 2)}
\]

\[
\text{Compare with:}
\]

\[
\text{(define a (list 1 2))}
\]

\[
\text{(define b (list 1 2))}
\]

\[
\text{(set-car! a 10)}
\]

\[
\text{b \implies (1 2)}
\]
Example 2: Pair/List Mutation

\[
\text{(define } x \text{ (list } 'a 'b) \text{)} \quad x \\
\]

- How mutate to achieve the result at right?

\[
\text{(set-car! (cdr } x \text{) (list } 1 2\text{))} \\
\]

1. Eval \((\text{cdr } x)\) to get a pair object
2. Change car pointer of that pair object

How do we know Table1 is an ADT implementation

- Potential reasons:
  - Because it has a type tag \(\text{No}\)
  - Because it has a constructor \(\text{No}\)
  - Because it has mutators and accessors \(\text{No}\)

- Actual reason:
  - Because the rest of the program does not apply any functions to Table1 objects other than the functions specified in the Table ADT
  - For example, no car, cdr, map, filter done to tables
  - The implementation (as an Alist) is hidden from the rest of the program, so it can be changed easily

Types for table1

- Here is everything the rest of the program knows

\[
\begin{align*}
\text{Table1}<k,v> & \quad \text{opaque type} \\
\text{make-table1} & \quad \text{void } \rightarrow \text{Table1}<\text{anytype,anytype}> \\
\text{table1-put!} & \quad \text{Table1}<k,v>, k, v \rightarrow \text{undef} \\
\text{table1-get} & \quad \text{Table1}<k,v>, k \rightarrow (v \mid \text{nil})
\end{align*}
\]

- Here is the hidden part, only the implementation knows it:

\[
\begin{align*}
\text{Table1}<k,v> & = \text{symbol } \times \text{Alist}<k,v> \\
\text{Alist}<k,v> & = \text{list}<k \times v>
\end{align*}
\]

Lessons so far

- Association list structure can represent the table ADT
- The data abstraction technique (constructors, accessors, etc) exists to support information hiding
- Information hiding is necessary for modularity
- Modularity is essential for software engineering
- Opaque type names denote information hiding

Information hiding in types: opaque names

- Opaque: type name that is defined but unspecified
- Given functions \(m1\) and \(m2\) and unspecified type \(\text{MyType}\):

\[
\begin{align*}
\text{(define } (m1 \text{ number}) \ldots) & \quad ; \text{number } \rightarrow \text{MyType} \\
\text{(define } (m2 \text{ myt}) \ldots) & \quad ; \text{MyType } \rightarrow \text{undef}
\end{align*}
\]

- Which of the following is OK? Which is a type mismatch?

\[
\begin{align*}
\text{(m2 (m1 } 10\text{))} & \quad ; \text{return type of } m1 \text{ matches} \quad ; \text{argument type of } m2 \\
\text{(car (m1 } 10\text{))} & \quad ; \text{return type of } m1 \text{ fails to match} \quad ; \text{argument type of } \text{car} \\
\text{car; pair}<A,B> & \rightarrow A
\end{align*}
\]

- Effect of an opaque name: no functions have the correct types except the functions of the ADT
Now let's talk about efficiency

• Speed of operations
  • put Fast
  • get Slow

• What if it's the Boston Yellow Pages?
  Really need to use other information to get to right place to search

Hash tables

• Suppose a program is written using Table1
• Suppose we measure that a lot of time is spent in `table1-get`
• Want to replace the implementation with a faster one
• Standard data structure for fast table lookup: hash table
• Idea:
  • keep N association lists instead of 1
  • choose which list to search using a hash function
  • given the key, hash function computes a number x where 0 <= x <= (N-1)
• Speed of hash table?

What's a hash function?

• Maps an input to a fixed length output (e.g. integer between 0 and N)
• Ideally the set of inputs is uniformly distributed over the output range
• Ideally the function is very rapid to compute
• Example:
  • First letter of last name:
    • 26 buckets
    • Non-uniform
  • Convert last name by position in alphabet, add, take modular arithmetic
    • GRIMSON: 7+18+9+13+19+15+14 = 95 (mod 26 = 17)
    • GREEN: 7+18+5+5+14=49 (mod 26 = 23)
• Uses:
  • Fast storage and retrieval of data
  • Hash functions that are hard to invert are very valuable in cryptography

Hash function output chooses a bucket

Store buckets using the vector ADT

- Vector: fixed size collection with indexed access
- Vector has constant speed access

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>make-vector</td>
<td>number, A → vector&lt;A&gt;</td>
</tr>
<tr>
<td>vector-ref</td>
<td>vector&lt;A&gt;, number → A</td>
</tr>
<tr>
<td>vector-set!</td>
<td>vector&lt;A&gt;, number, A → undef</td>
</tr>
</tbody>
</table>

The Bucket Abstraction

(define make-buckets N v) (make-vector N v)
(define make-buckets make-vector)
(define bucket-ref vector-ref)
(define bucket-set! vector-set!)
Table2: Table ADT implemented as hash table

(define t2-tag 'table2)
(define (make-table2 size hashfunc)
  (let ((buckets (make-buckets size nil)))
    (list t2-tag size hashfunc buckets)))

(define (size-of tbl) (cadr tbl))
(define (hashfunc-of tbl) (caddr tbl))
(define (buckets-of tbl) (cadddr tbl))

• For each function defined on this slide, is it
  • a constructor of the data abstraction?
  • an accessor of the data abstraction?
  • an operation of the data abstraction?
  • none of the above?

get in table2

(define (table2-get tbl key)
  (let ((index ((hashfunc-of tbl) key (size-of tbl))))
    (find-assoc key (bucket-ref (buckets-of tbl) index))))

• Same type as table1-get

put! in table2

(define (table2-put! tbl key val)
  (let ((index ((hashfunc-of tbl) key (size-of tbl)))
        (buckets (buckets-of tbl))
        (bucket-set! buckets index
          (add-assoc key val (bucket-ref buckets index)))))

• Same type as table1-put!

Table2 example

(define tt2 (make-table2 4 hash-a-point))
(table2-put! tt2 (make-point 5 5) 20)
(table2-put! tt2 (make-point 5 7) 15)
(table2-get tt2 (make-point 5 5))

Is Table1 or Table2 better?

• Answer: it depends!
  • Table1: make extremely fast
    put! extremely fast
    get O(n) where n=# calls to put!
  • Table2: make space N where N=specified size
    put! must compute hash function
    get compute hash function plus O(n)
    where n=average length of a bucket

• Table1 better if almost no gets or if table is small
• Table2 challenges: predicting size, choosing a hash function
  that spreads keys evenly to the buckets

Summary

• Introduced three useful data structures
  • association lists
  • vectors
  • hash tables
• Operations not listed in the ADT specification are internal
• The goal of the ADT methodology is to hide information
• Information hiding is denoted by opaque type names