OOPLs

Historical roots:

CLU language (B.Liskov and J.Guttag, "Abstraction and Specification in Program Development", out of print but copy on reserve in Math Library)

- Encapsulation
- Specification of abstract datatypes
  - *requires, modifies, effects*
- Mutability
- Equality checking

Data Abstraction and OOPLs

- Abstraction interface
  - Mutators, Observers, Constructors
  - Abstraction function
  - Representation invariant
- Iterators - C++ and Java examples

- Dynamic dispatch
  - Overriding in inheritance hierarchy
  - Overloading
  - Efficient implementation strategies

Data Abstraction

- Can use any internal representation for storing queues as long as can make it behave like a queue.
- Interface to the queue data abstraction is same, no matter what the *rep type*.
  - Knowledge of interface is sufficient to use this queue code; (centralized dependence)
  - Users can't change the abstraction unless allowed by interface.
  - Can change *rep type* for efficiency without disturbing users of the abstraction

Example - Queue

- Type: first in, first out storage discipline
- Operations:
  - *enqueue(q,x)* - adds x onto queue q
  - *qnull(q)* - returns boolean check if q is empty
  - *qhd(q)* - selects front element of queue q
  - *dequeue(q)* - yields queue obtained by removing front element of queue q
  - *Qerror* - exception raised by *qhd* or *dequeue* applied to an empty queue

Possible Implementations

- **Using 'a-list**

enqueue(q,x) = q @ [x]; (*costly, sum of lengths of 2 lists*)
dequeue(x::q) = q (*cheap*)| dequeue nil = raise Qerror;

- **Using user-defined datatype**

datatype 'a queue = <u>empty</u> | <u>enqueue</u> of 'a queue * 'a
fun dequeue (enqueue (empty,x)) = empty |
fun dequeue (enqueue(q,x)) = enqueue ((dequeue q), x) |
fun dequeue (empty) = raise Qerror;

- **Using 2 'a-lists (one for adding and one for removing and then have to switch when run out of removing list)**

datatype 'a queue = Queue of ('a list * 'a list)
normal form for this representation is maintained by function *norm* which has to be called after every removal of an element.
fun norm (Queue ([ ],tail) = Queue ((reverse tail),[ ]) | norm q = q;

Specification

- **queue is a data abstraction containing integers following a *first in, first out* discipline.**
- **Implementation separated from specification**
- **Operation described in terms of its type signature, what it modifies, what it requires as a precondition and its effect**
  - **For templates (generics) allows use of type parameter**

CLU Specification

- *Requires* = constraints on the use of an operation, if any
- *Modifies* = side effects on inputs
- *Effects* = defines operation behavior on allowed inputs

Operations Specification

enqueue = proc (q:queue, x: int) returns (queue)
  modifies: q
  effects: Adds x to q
dequeue = proc (q:queue) returns (queue)
  modifies: q
  requires: q be nonempty
  effects: Returns q with one less element.
qhd = proc (q:queue) returns (int)
  effects: Returns element at head of q
  requires: q be nonempty.
Qnull = proc (q:queue) returns (bool)
  effects: Returns true if q is empty, else false.

Mutability

- **Mutable data abstractions have values which can *change* during execution.**
  - **Used to model real-world entities**
  - **Tricky to manage for shared objects**
  - **Destructive operations are performed; more space efficient**
- **Mutability is property of the abstract data type, NOT the implementation**
  - **Mutable types need mutable rep types**
  - **Immutable types can use mutable or immutable rep types**

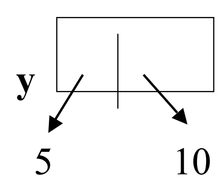
Mutability

- **Immutable data abstractions are *assign-once* variables**
  - **E.g., integers, points in a plane**
  - **Safer for shared objects**
  - **Operations on this type return new object of the type with altered values.**
  - **Creates need for garbage collection**

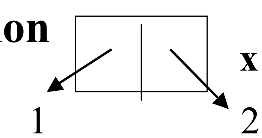
Classes of Operations

- **Constructors**
  - **Create objects of a datatype**
- **Mutators**
  - **Modify objects of a datatype -** ***enqueue, dequeue***
- **Observers**
  - **Given object of a datatype, return values related to that object -** ***qnull, qhd***

Equality Checking



- **Need to provide in the interface**
- **Can use a canonical representation**



  - **E.g., rationals, R = Rat of int * int;**

  fun make (a,b:int) = Rat (a,b).

  Then val x=make(1,2); val y=make(5,10); x=y isn't true!

  However, the following works:

  make2(a,b)=(Rat(a div gcd(a,b), b div gcd(a,b)));
- **Can also create own equality function within the abstraction**

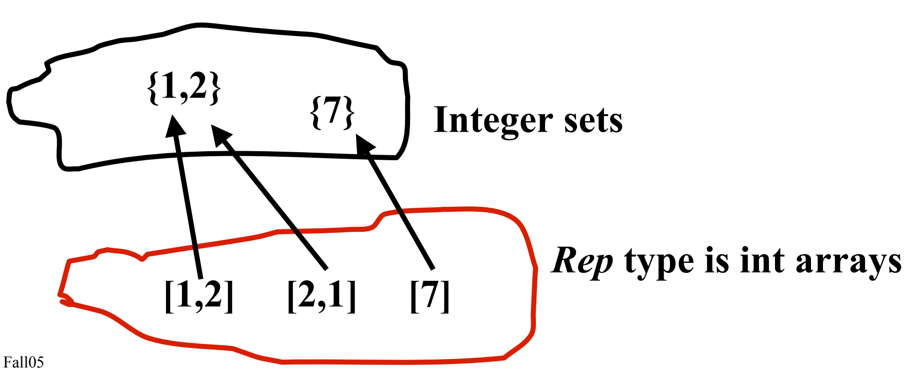
  Eg., fun equalrat(Rat(a,b),Rat(c,d)) = (a*d = c*b)

Abstract Datatype

- Can refer to abstract datatype and its *rep* type separately
- Can refer to the mappings between these 2 worlds
  – *Abstraction Function:* maps a *rep* object to its corresponding abstract datatype object; defines meaning of the representation
  – *Representation Invariant:* statement of a property that all legitimate *reps* of abstract objects satisfy

Abstraction Function

- More than 1 *rep* value may represent same abstract value
  – Integer sets represented in arrays
  [1,2] and [2,1] both are array *reps* of {1,2}



Representation Invariant

- Think about (x,y) coordinates represented by polar coordinates (length,angle).

  g(r) = (r.ln *cos(r.ang), r.ln * sin(r.ang))

  Then Invar(r)= (r.ln>0 and 0<= r.ang <=2π) or

  (r.ln = 0 and r.ang = 0)
- For int sets represented as an int array R,

Invar(R) = for all k,j, low (R) <= k < j <= high(R)

and R[k] != R[j] (since sets have no multiple elements)

CLU Generic Functions

Search function on character data:

search = proc (v:char, b: array of char) returns (x:bool)

requires: b sorted in non-decreasing order

effects: true returned iff b[j]=v for some j

Generic search function:

search = proc [t:type](v:t, a: array[t]) returns (int)

requires: t has operations *equal, lt*: proctype (t,t) returns (bool) such that t is totally ordered by *lt*, and a is sorted in ascending order based on *lt*

effects: if v is in a, returns j such that a[j]=v; otherwise, returns high(a)+1 (i.e., upper bnd on a +1)

Iterators

- **If abstract datatype is a collection of objects, you may want to examine each object in the collection**
- **How to accomplish this?**
  - **Write a function in the interface that extracts the objects, 1 by 1, performs some calculation on them and then recreates the collection**
  - **Copy the objects in the collection to an immutable type (like sequence in CLU). Return that to the user to use**

Iterators

- **Provide a special function for the abstract datatype: an iterator**

  elements = iter (s:intset) yields (int)

  ***requires:* s not be modified by calling loop body (or consequences can't be determined)**

  ***effects:* yields elements of s one by one in arbitrary order**
- **Iterators can be nested**
  - **They operate as though each has its own copy of the collection.**

Enumerations in Java

- **Java - Enumeration object keeps copy of collection or a copy of a reference to it**
  - **Affects whether or not changing the collection while iterating disturbs the enumeration**
  - **Use *polymorphic* container class and then downcast to proper object type**
    - **e.g., SetEnumeration returns Object type; needs to be cast to actual type at each use**
  - **Enumeration is a Java *interface* with standard functions that classes which implement it must provide**

C++ Iterator Example

```
class stack { private: elt *s; int top; friend class stack_iter;
    const int EMPTY = -1;
    public: stack(){s = new elt[100]; top = -1;} …}
class stack_iter{//will enumerate stack from bottom to top of stack
    private: elt *st; int n; int t;
    //invariant: elements in st[0..n] have already been returned
    stack_iter(stack &goOver){ // creates copy of stack
        t = goOver.top;
        st = new elt[t+1];
        for (int j=0; j<=t; ++j)
            st[ j]=goOver.s[j];
        n = goOver.EMPTY;} //initializes subscript pointing into copy
    boolean getNext(elt &val){
        if (n < t) {val = st[++n]; return 1;} else return 0;
    } }
```

Iterators in C^{++}

- Can’t define iterator as subclass of the collection class
  - Because then each iterator could only work with respect to one collection object
- Can’t define iterator as member of the collection class
  - Because member functions have no way to preserve state between calls (class vars are not enough since they are shared by all objects)

Iterators in C^{++}

- There is NO natural subtyping relation between iterators and the collections they iterate over!
- Solution - break encapsulation to create an iterator
  - Use *friend* methods which lets iterator see into the private collection instance variables