Data Abstraction, Types, and Polymorphism

February 25, 2003

Project 2

• Due Friday, February 28

• Clarifications on news group
  – Don’t need to test handling of erroneous clients
    • Passing in null as parameter
    • newVertex twice with same name
    • Modifying graph during iteration
    • Using a Vertex after removing it
  – Do need to test GraphFactory
    • But can limit to checks of form read(write(g)) = g

Project 2

• Make sure project 2 compiles!
  – You won’t get points for code that doesn’t compile
    • In project 1, we had some weird glitches with line endings, or the lack thereof
    • Mixes of \n and \r in one file
    • Lines joined with no line separators

Data Abstraction

• Data abstraction = objects + operations
  – List + { addFirst, addLast, removeFirst, ... }
  – Set + { add, contains, ... }

• Categories of operations
  – Constructors (creators/produces)
  – Mutators
  – Observers
Abstraction Function

• Specification for data structure is abstract
• Implementation of data structure is concrete
• How do you know if implementation meets the spec?

• Abstraction function: concrete → abstract
  – Relates implementation to abstraction

Example

class IntSet { int [] elts; ... }

  – \( AF(s) = \{ s.e[0] \mid 0 \leq i < s.e.length \} \)

• You always need an abstraction function when you build a data abstraction
  – Often it’s implicit

Representation Invariant(s)

• Properties of data structure that must always hold
  – After the constructor has finished
  – Before and after each operation
  – E.g., binary search tree property

• Part of the (internal) specification

• Be careful of exposing the rep
  – Dangerous, because rep invariant may be violated
  – Be wary of returning references to internal mutable structures
    * e.g., returning a pointer to an internal array

Methods Inherited from Object

• equals
  – Override with appropriate method

• hashCode
  – Always override if you override equals
    * two equal objects must have equal hashCodes

• toString
  – Always override; try to provide maximal information

• clone
  – Hard to use correctly; avoid if possible
Mutability

- Many data abstractions are mutable
  - Operations can change internal state

- Things to watch out for
  - Mutating shared data
  - Mutating an object in a container

- Benevolent side effects
  - Side effects that are not visible to client (e.g., cache)
  - Not "conceptual" mutability
    - Don’t change abstract value

Type Hierarchy

- Recall: A class A can extend other classes B and implement interfaces C
  - A, B, and C are all types
  - A is a subtype of B, and A is a subtype of C

- Substitution (or subsumption):
  - If B is a subtype of A, then a B can be used anywhere an A is expected.

The Meaning of Subtypes

- Method signatures must match exactly to override
  - Parameter and return types
  - Overriding method can throw fewer exceptions

- This is all checked at compile time

- Also important: behavior of overriding method must match

Overriding/Implementing Methods

- Given
  ```
  /** Search array for value */
  /** @precondition: d is sorted */
  /** @postcondition: returns index i s.t. d[i] == value,
  * or -1 if no such value exists */
  int find(int[] d, int value);
  ```

- Can we implement find() with
  a) A method that accepts any array?
  b) A method that requires d is sorted and there exists i s.t. d[i] = value?
  c) A method that either returns -1 or the first index i s.t. d[i] = value?
  d) A method that it throws "NoSuchElementException" rather than returning -1 when value does not occur in d?

- Summary: When implementing a spec, we can have
  - Weaker preconditions (more possible call states allowed)
  - Stronger postconditions (fewer possible return states)
Polymorphism Using Object

class IntegerStack {  
class Entry {  
  Integer elt; Entry next;  
  Entry(Integer i, Entry n) { elt = i; next = n; }  
}  
Entry theStack;  
void push(Integer i) {  
  theStack = new Entry(i, theStack);  
}  
Integer pop() throws EmptyStackException {  
  if (theStack == null)  
    throw new EmptyStackException();  
  else {  
    Integer i = theStack.elt;  
    theStack = theStack.next;  
    return i;  
  }  
}
}

IntegerStack Client

IntegerStack is = new IntegerStack();  
Integer i;  
is.push(new Integer(3));  
is.push(new Integer(4));  
i = is.pop();

• This is OK, but what if we want other kinds of stacks?  
  – Need to make one XStack for each kind of X  
  – Problems: Not pretty, code bloat, maintainability nightmare

Polymorphism Using Object

class Stack {  
class Entry {  
  Object elt; Entry next;  
  Entry(Object i, Entry n) { elt = i; next = n; }  
}  
Entry theStack;  
void push(Object i) {  
  theStack = new Entry(i, theStack);  
}  
Object pop() throws EmptyStackException {  
  if (theStack == null)  
    throw new EmptyStackException();  
  else {  
    Object i = theStack.elt;  
    theStack = theStack.next;  
    return i;  
  }  
}
}

Stack Client

Stack is = new Stack();  
Integer i;  
is.push(new Integer(3));  
is.push(new Integer(4));  
i = (Integer) is.pop();

• Now Stacks are reusable  
  – push() works the same  
  – But now pop() returns an Object  
    • Have to downcast back to Integer  
    • Not checked until run-time
**General Problem**

• When we move from an X container to an Object container
  – Methods that take X’s as input parameters are OK
    • If you’re allowed to pass Object in, you can pass any X in
  – Methods that return X’s as results require downcasts
    • You only get Objects out, which you need to cast down to X
• This is a general feature of *subtype* polymorphism

**Parametric Polymorphism**

• Idea: We can *parameterize* the Stack class by its element type

• Syntax:
  – Class declaration: `class A<T> { ... }
  • A is the class name, as before
  • T is a type variable, can be used in body of class (...)  
  – Usage: `A<Integer> x;`
    • We instantiate A with the Integer type

**Parametric Polymorphism for Stack**

```java
class Stack<Element> {
    class Entry {
        Element elt; Entry next;
        Entry(Element i, Entry n) { elt = i; next = n; }
    }
    Entry theStack;
    void push(Element i) {
        theStack = new Entry(i, theStack);
    }
    Element pop() throws EmptyStackException {
        if (theStack == null)
            throw new EmptyStackException();
        else {
            Element i = theStack.elt;
            theStack = theStack.next;
            return i;
        }
    }
}
```

**Stack<Element> Client**

```java
Stack<Integer> is = new Stack<Integer>();
Integer i;
is.push(new Integer(3));
is.push(new Integer(4));
i = is.pop();
```

• No downcasts
• Type-checked at compile time
• No need to duplicate Stack code for every usage
### The Identity Function

- Suppose B is a subtype of A
  1. static A id(A x) { return x; }
  2. static A id(B x) { return x; }
  3. static B id(A x) { return x; }
  4. static B id(B x) { return x; }

- Can’t pass an A to 2 or 4
- 3 doesn’t type check
- Can pass a B to 1 but you get an A out

### Parametric Polymorphism, Again

- Observation: id() doesn’t care about the type of x
  - It works for any type

  ```java
  static <T> T id(T x) { return x; }
  ```

- So use parametric polymorphism:

  ```java
  Integer i = id(new Integer(3));     // Notice no need to
  // instantiate id; compiler
  // figures it out
  ```

### Parametric Polymorphism in Java

- Slated to be part of Java 1.5
  - Available in pre-release form now
  - Called "generics"

- Available now
  - In pre-release form
    - linuxlab/~pugh/adding_generics-1.3-en.zip

### gjc

- gj compiler installed on linuxlab
  - Available as ~pugh/bin/gjc
  - Can add ~pugh/bin to your path

- gj translates Java w/parametric polymorphism into standard Java byte codes
  - Intuitively, compiler translates gj to Java
  - Compiled gj programs are valid Java, can be run on any correct implementation of JVM
**gjc Libraries**

- Comes with replacement for java.util.*
  - class LinkedList<A> { ... }
  - class HashMap<A, B> { ... }
  - interface Collection<A> { ... }
  - interface Comparable<A> { ... } // in java.lang

**gj Translation via Erasure**

- (According to OOPSLA98 paper)
- gj replaces uses of type variables with Object
  - class A<T> { ...T x;... } ==> class A { ...Object x;... }
- Adds downcasts wherever necessary
  - Integer x = A<Integer>.get(); ==> Integer x = (Integer) (A.get());
- Some complications with overloading
- Need to be careful with security
  - LinkedList<SecureChannel>

**Limitations of gj Translation**

- Some type information not available at run-time
  - Recall type variables T are rewritten to Object
- Disallowed, assuming T is type variable
  - new T() would translate to new Object() (gjc error)
  - new T[n] would translate to new Object[n] (gjc warn)
    - Use public static <A>[] newInstance(A[] a, int n) in java.lang.reflect.Array
    - Some casts/instanceofs that use T
      - (Only ones the compiler can figure out are allowed)

**Using gj with Legacy Code**

- gj translates via type erasure
  - class A<T> ==> class A
- Thus class A is available as a “raw type”
  - class A<T> { ... }
  - class B { A x; }
- Sometimes useful with legacy code, but...
- Dangerous feature to use, plus unsafe
  - Relies on implementation of generics, not semantics
<table>
<thead>
<tr>
<th>Summary: Kinds of Polymorphism</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Subtype polymorphism</td>
</tr>
<tr>
<td>– Use subtype wherever supertype allowed</td>
</tr>
<tr>
<td>• Parametric polymorphism</td>
</tr>
<tr>
<td>– When classes/methods work for any type; uses type variables</td>
</tr>
<tr>
<td>• Ad-hoc polymorphism</td>
</tr>
<tr>
<td>– Method overloading in Java</td>
</tr>
<tr>
<td>– + on ints, doubles and Strings</td>
</tr>
</tbody>
</table>