rep invariants, equality, visitors

Daniel Jackson
October 20, 2008
plan for today

recall strategy for avoiding bugs

- make them impossible
- don’t insert them
- make them easy to find

topics

- advice on implementing types
- equality and how to code it
- rep invariants & how to exploit them

patterns

- Iterator and Visitor
advice on implementing types
step 1: design a rep

desiderata
• easy to program (and get right!)
• good enough performance

usually
• a couple of fields of existing types suffices
• so before inventing a complex type, check Java collections and your own

sometimes
• a tricky structure or algorithm is needed
• first, see if someone’s done it before (eg, look it up in CLR book)

always
• write a rep invariant to clarify the design
step 2: write the methods

required methods first
  \· from \texttt{Object} class: \texttt{equals}, \texttt{hashCode}, \texttt{toString}
  \· from any interface the class implements
  \· when overriding, mark with \texttt{@Override}

constructors
  \· for an immutable type, some private constructors often help

producers (return new values of type) and observers (return other types)
  \· whenever possible, build on each other
  \· separate core methods (eg, \texttt{size}) from those that sit on top (eg. \texttt{isEmpty})

incomplete methods
  \· use \texttt{UnsupportedOperationException} to get it to compile

more on \texttt{hashCode} in part 3 of course
step 3: rep invariant

code the rep invariant
  · as a `checkRep` method
  · for immutables, call it at the end of all constructors

as you write the operations
  · ask yourself `why` they preserve the rep invariant
step 4: assertions and tests

runtime assertions

\· are your friend: use them freely
\· turn on by adding \texttt{-ea} to VM args in Eclipse

write JUnit test suite for your class

\· will help you find bugs earlier, and make debugging easier
\· take the trouble to write a \texttt{toString} that produces helpful output
equality: basics
objects often used as keys

- need to compare them
- eg, `Literal` used as key in `Environment`

Java convention

- the class `Object` has a method that every class inherits
  
  `Object.equals: Object -> boolean`

- by convention, this method is used to compare objects for equality

- collections especially assume this: call `equals` on keys

- the inherited method is usually wrong for immutable types

- so must override by explicitly declaring a method
  
  `MyType.equals: Object -> boolean`
why inherited equality fails

the problem

• `Object.equals` compares objects with `==`
• this makes any two distinct objects unequal
• even if they have the same value

example

• the “same” pairs are unequal:

```java
public class Pair {
    private int fst, snd;
    public Pair (int f, int s) {fst=f; snd=s;}

    public static void main (String[] args) {
        Pair p1 = new Pair (1, 2);
        Pair p2 = new Pair (1, 2);
        System.out.println (p1 == p2 ? "yes" : "no");
        System.out.println (p1.equals(p2) ? "yes" : "no");
    }
}
```
standard equals method

correct code for Pair.equals

@Override
public boolean equals (Object that) {
    if (this == that) return true;
    if (!(that instanceof Pair)) return false;
    Pair p = (Pair) that;
    return p.fst == fst && p.snd == snd;
}

remember: comparison is with any object reference

· need to check type of arg, and whether null
· you may be tempted to write this, but don’t: it will just overload equals
  public boolean equals (Pair that) {...}
· write @Override and compiler will catch the bug
a design puzzle

interning objects

' suppose you have a structure containing objects of type C
' you want to intern them: that is, have one object for each value
' so you write this code, but it won't work (why not?)

```java
public class C {
    private String s;
    public static Map<C,C> allocated = new ListMap<C,C>();
    public C intern () {
        C c = allocated.get(this);
        if (c == null) {
            allocated = allocated.put(this, this);
            return this;
        }
        return c;
    }
}
```
approaches

the problem: one equals method

- if it compares references with ==, then lookup won't find match
- it it compares values, then interning is pointless!

have collection take equality predicate as argument

- can't use standard Java collections: will have to make your own
- but see use of comparator objects in ordered types like `java.util.TreeSet`

use component as key instead of whole object

- eg, allocated maps `String` to `C`
- this is how the factory method of `PosLiteral` works (previous lecture)

for key, make wrapper around `C` object with its own `equals`

- not terrible, but a bit ugly
rep invariants
rep invariant $R$

- defines set of legal representation values
- documented and implemented as checkRep

abstraction function $A$

- interprets legal representation values as abstract values
- documented and implemented as toString
how to establish invariants

for state machines
• establish invariant in initial state
• ensure that all transitions preserve invariant

for mutable types, the same approach
• a mutable object *is* a state machine

for immutable types, a similar story
• objects can’t change
• assume any argument you’re given satisfies the invariant
• ensure any result you create satisfies it too

who gets to preserve the invariant?
• by hiding the rep, can limit to the methods of the ADT itself
implications

**a strong invariant means**

- methods can assume more about arguments
- allows checks to be omitted and optimizations to be applied
- but methods must do more to ensure results satisfy invariant

**rep design = rep invariant**

- the choice of rep invariant characterizes the design of the rep!
common invariants

these invariants
\- are commonly used
\- provide concrete benefits

examples
\- **no nulls**: no need to check before calling method
\- **acyclic**: no need to worry about looping
\- **ordered**: can navigate efficiently; can stop when key value is passed
\- **no duplicates**: can stop when find first match
\- **caching**: can do fast look up
example: invariant for Clause
writing the invariant

rep invariant for **Clause** written as executable method

```java
public class Clause {
    private final List<Literal> literals;
    static final boolean CHECKREP = true;
    void checkRep () {checkRep (literals);}
    void checkRep (List<Literal> ls) {
        assert ls != null : "Clause, invariant: literals non-null";
        if (!ls.isEmpty()) {
            Literal first = ls.first(); List<Literal> rest = ls.rest();
            assert first != null : "Clause, invariant: no null elements";
            assert !rest.contains(first) : "Clause, invariant: no duplicates";
            assert !rest.contains(first.getNegation()) : "Clause, invariant: no literal and its negation";
            checkRep (rest);
        }
    }
    private Clause(List<Literal> literals) {
        this.literals = literals;
        if (CHECKREP) checkRep();
    }
}
```

what's the computational cost of checkRep?
exploiting the invariant

an equals method for Clause

```java
@Override
public boolean equals (Object that) {
    if (this == that) return true;
    if (!(that instanceof Clause)) return false;
    Clause c = (Clause) that;
    if (size() != c.size()) return false;
    for (Literal l: literals)
        if (!c.contains(l)) return false;
    return true;
}
```

how invariant is exploited

• since literals is non-null, can use in for-loop without null check
  implicit call to literals.iterator will throw exception if literals is null
• since no duplicate literals, can check containment in one direction only
  that is, given two sets S and T:  \( S = T \iff |S| = |T| \land S \subseteq T \)
preserving the invariant

no free lunch

• you have to do some work to establish the invariant

element: Clause.add

/**
 * Add a literal to this clause; if already contains the literal's negation, return null
 * Requires: l is non-null
 * @return the new clause with the literal added, or null
 */
public Clause add(Literal l) {
    if (literals.contains(l)) return this;
    if (literals.contains(l.getNegation())) return null;
    return new Clause(literals.add(l));
}

• what impact does each part of the invariant have?
exploiting the invariant

exercise: how does reduce exploit the invariant?

/**
 * Requires: literal is non-null
 * @return clause obtained by setting literal to true
 * or null if the entire clause becomes true
 */
public Clause reduce(Literal literal) {
    List<Literal> reducedLiterals = reduce(literals, literal);
    if (reducedLiterals == null) return null;
    else return new Clause(reducedLiterals);
}

private static List<Literal> reduce(List<Literal> literals, Literal l) {
    if (literals.isEmpty()) return literals;
    Literal first = literals.first();
    List<Literal> rest = literals.rest();
    if (first.equals(l)) return null;
    else if (first.equals(l.getNegation())) return rest;
    else {
        List<Literal> restR = reduce(rest, l);
        if (restR == null) return null;
        return restR.add(first);
    }
}
iterator pattern
iteration in Java

recall how our solver found a minimal clause

iterate over clauses

```
Clause min = null;
for (Clause c : clauses) {
    if (c.isEmpty()) return null;
    if (min == null || c.size() < min.size()) min = c;
}
...
```

how does this work?

hidden iterator at play
the iterator pattern

a Java shorthand

' the statement
    for (E e: c) {...}
' is short for
    Iterator i = c.iterator();
    while (i.hasNext()) {
        E e = i.next();
        ...
    }

iterator interface

public interface Iterator<E> {
    boolean hasNext();
    E next();
    void remove();
}

list iterator

public class ListIterator<E> implements Iterator<E> {
    List<E> remaining;
    public ListIterator (List<E> list) {
        remaining = list;
    }
    public boolean hasNext () {
        return !remaining.isEmpty();
    }
    public E next () {
        E first = remaining.first();
        remaining = remaining.rest();
        return first;
    }
}

iterator method

public abstract class List<E> implements Iterable<E> {
    public Iterator<E> iterator () {
        return new ListIterator<E>(this);
    }
}
why a stateful object in a side-effect free program?

- the only convenient way to do iteration in Java
- so long as iterator used only in for loop as shown, no mutability issues arise
visitor pattern
localizing functions

Interpreter pattern: look what we’re doing

• declare function over datatype
  
  \[ \text{size: List}\langle T \rangle \to \text{int} \quad \text{where} \quad \text{List}\langle T \rangle = \text{Empty} + \text{Cons}\left( \text{first: } T, \text{ rest: List}\langle T \rangle \right) \]

• break function into cases, one per variant
  
  \[
  \begin{align*}
  \text{size (Empty)} &= 0 \\
  \text{size (Cons(first: e, rest: l))} &= 1 + \text{size(l)}
  \end{align*}
  \]

• but then split cases across classes! can’t we keep them together?

• in functional language can do exactly this: (in ML, eg)
  
  \[
  \text{fun size Empty = 0} \\
  \text{\quad | Cons(e, l) = 1 + size(l)}
  \]

solution: localize function definition in “visitor”

• hard to grasp first time, but easy once you know the pattern

• a useful and common idiom, esp. for compilers

• good check of your understanding of dynamic dispatch & overloading
basic visitor structure

visitor

```java
public interface ListIntVisitor<E> {
    int onEmpty (Empty<E> l);
    int onCons (Cons<E> l);
}
public class SizeVisitor<E> implements ListIntVisitor<E>{
    public int onEmpty(Empty<E> l) {return 0;}
    public int onCons(Cons<E> l) {return 1 + l.rest().accept(this);}
}
```

datatype and variants

```java
public abstract class List<E> {
    public abstract int accept(ListIntVisitor<E> visitor);
}
public class Empty<E> extends List<E> {
    public int accept(ListIntVisitor visitor) {return visitor.onEmpty(this);}
}
public class Cons<E> extends List<E> {
    public int accept(ListIntVisitor<E> visitor) {return visitor.onCons(this);}
}
```

usage

```java
int size = myList.accept(new SizeVisitor<E>());
```
the visitor carousel

<table>
<thead>
<tr>
<th>Step</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>l.accept(v)</td>
</tr>
<tr>
<td>2</td>
<td>v.onCons(l)</td>
</tr>
<tr>
<td>3</td>
<td>l.rest().accept(v) + 1</td>
</tr>
<tr>
<td>4</td>
<td>v.onEmpty(e)</td>
</tr>
<tr>
<td>5</td>
<td>return 0</td>
</tr>
</tbody>
</table>

note how

- control passes back and forth between visitor and datatype objects
- function is computed at visitor (steps 3 and 5)
going polymorphic

accept methods only work for visitor that returns integer

```java
going polymorphic

```java
public interface ListIntVisitor<E> {  
  int onEmpty (Empty<E> l);  
  int onCons (Cons<E> l);  
}

so make the visitor polymorphic

' new interface

```java
public interface ListVisitor<E,T> {  
  T onEmpty (Empty<E> l);  
  T onCons (Cons<E> l);  
}

' new accept methods

```java
public <T> T accept(ListVisitor<E,T> visitor) {return visitor.onEmptyList(this);}

' new visitor

```java
public class SizeVisitor<E> implements ListVisitor<E,Integer>{  
  public Integer onEmpty(Empty<E> l) {return 0;}  
  public Integer onCons(Cons<E> l) {return 1 + l.rest().accept(this);}
}
```
final refinement

accept method is almost boilerplate

```java
public class Cons<E> extends List<E> {
    public int accept(ListIntVisitor<E> visitor) {
        return visitor.onCons(this);
    }
}
```

can make identical by exploiting overloading

‣ new interface

```java
public interface ListVisitor<E,T> {
    T visit (Empty<E> l);
    T visit (Cons<E> l);
}
```

‣ new accept method: same in all variants

```java
public <T> T accept(ListVisitor<E,T> visitor) {
    return visitor.visit(this);
}
```

‣ new visitor

```java
public class SizeVisitor<E> implements ListVisitor<E,Integer>{
    public Integer visit (Empty<E> l) {
        return 0;
    }
    public Integer visit (Cons<E> l) {
        return 1 + l.rest().accept(this);
    }
}
```
summary
principles

use rep invariants to prevent bugs
  · and to make them easier to find
  · design of type = rep invariant

equality is tricky
  · for immutables, compare contents not object refs
  · (not covered in lecture) if you override equals, must override hashCode too

visitor pattern
  · some boilerplate code in datatypes
  · allows one function/class