rep invariants, equality
using sets

recall computing set of vars appearing in a formula

- declare function
  \[ \text{vars}: F \rightarrow \text{Set}<\text{Var}> \]

- declare datatype
  \[ F = \text{Var(name: String)} + \text{Or(left:F,right:F)} + \text{And(left:F,right:F)} + \text{Not(formula:F)} \]

- define function over variants
  \[ \text{vars} (\text{Var}(n)) = \{\text{Var}(n)\} \]
  \[ \text{vars} (\text{Or}(fl, fr)) = \text{vars}(fl) \cup \text{vars}(fr) \]
  \[ \text{vars} (\text{And}(fl, fr)) = \text{vars}(fl) \cup \text{vars}(fr) \]
  \[ \text{vars} (\text{Not}(f)) = \text{vars}(f) \]

where do sets come from?

- defined structurally like this
  \[ \text{Set}<T> = \text{List}<T> \]

- but should be defined by \textit{operations} instead: \{\}, \cup
a new viewpoint

congcre datatypes
  • datatypes defined by their structure or representation

abstract datatypes
  • datatypes defined by their operations or behavior

extending the type repertoire
  • used to thinking of basic types behaviourally:
    integers: +, *, <, =
    array: get(a,i), store(a,i,e)
  • abstract datatypes: user-defined types
    string: concat(s,t), charAt(s,i)
    set: {}, ∪, ∈
public interface Set<E> {

    public Set<E> add (E e);

    public Set<E> remove (E e);

    public Set<E> addAll (Set<E> s);

    public boolean contains (E e);

    public E choose ();

    public boolean isEmpty ();

    public int size ();

}
a set implementation

public class ListSet<E> implements Set<E> {
    private List<E> elements;

    public ListSet () {elements = new EmptyList<E> ();} 

    public Set<E> add (E e) {
        if (elements.contains (e)) return this;
        return new ListSet<E> (elements.add (e));
    }

    public Set<E> remove (E e) {
        if (isEmpty()) return this;
        E first = elements.first();
        ListSet<E> rest = new ListSet<E> (elements.rest());
        if (first.equals(e))
            return rest;
        else
            return rest.remove(e).add(first);
    }

    public boolean contains (E e) {
        return elements.contains(e);
    }

    ...
}
what makes an abstract type?

defined by operations

- an integer is something you can add, multiply, etc
- a set is something you can test membership in, union, etc

representation is hidden or “encapsulated”

- client can’t see how the type is represented in memory
- is integer twos-complement? big or little endian?
- is set a list? a binary tree? an array?

language support for data abstraction

- packaging operations with representations
- hiding representation from clients
two reasons for encapsulation of representations

rep independence

• if client can’t see choice of rep, implementor can change it
  • eg: integers: your program can run on a different platform
  • eg: sets: programmer can switch rep from list to array

rep invariants

• not all values of the rep make legal abstract values
• prevent client from accessing rep so code of ADT can preserve invariants
  • eg: sets: make sure element does not appear twice
domain specific and generic types

- some types are specific to a domain (clause, literal)
- some have wide application (list, set)
- widely applicable types are usually polymorphic
- these are the “classic ADTs” found in the standard package `java.util`

often called “Java collection framework” in Java

© Daniel Jackson 2008
# a zoo of types

<table>
<thead>
<tr>
<th>type</th>
<th>overview</th>
<th>producers</th>
<th>observers</th>
<th>common reps</th>
</tr>
</thead>
<tbody>
<tr>
<td>list</td>
<td>sequence for</td>
<td>add, append</td>
<td>first, rest, ith</td>
<td>array, linked list</td>
</tr>
<tr>
<td>queue</td>
<td>concatenation and front-</td>
<td>enq, deq</td>
<td>first</td>
<td>array, list, circular buffer</td>
</tr>
<tr>
<td>stack</td>
<td>append</td>
<td>push, pop</td>
<td>top</td>
<td>array, list</td>
</tr>
<tr>
<td>map</td>
<td>associates keys and values</td>
<td>put</td>
<td>get</td>
<td>association list, hash table, tree</td>
</tr>
<tr>
<td>set</td>
<td>unordered collection</td>
<td>insert, remove</td>
<td>contains</td>
<td>map, list, array, bitvector, tree</td>
</tr>
<tr>
<td>bag</td>
<td>like set, but element can appear more than once</td>
<td>insert, remove</td>
<td>count</td>
<td>map, array, association list</td>
</tr>
</tbody>
</table>
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 *Object* class: *equals*, *hashCode*, *toString*
› from any interface the class implements
› when overriding, mark with *@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, *size*) from those that sit on top (eg, *isEmpty*)

incomplete methods

› use *UnsupportedOperationException* to get it to compile
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 `-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 `toString` that produces helpful output
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
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

eexample

‣ 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");
    }
}
```
correct code for `Pair.equals`

- compare the fields

```java
@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

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

the problem: one equals method

- if it compares references with ==, then lookup won’t find match
- if it compares values, then interning is pointless!
  
  The whole point of interning is to have a cheap equals method

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
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();
    }
    ...
}
```

flag to turn expensive check off

messages give invariant informally

check rep for each constructed value

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(lrl 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

\[
\text{for (E e: c) {...}}
\]

\[=\]

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