6.005
elements of
software
construction

analyzing data types, part 2

Daniel Jackson

october 22, 2007
topics for today

introduction
  • reasoning vs. testing

review of key concepts
  • rep invariants, abstraction function, pre/post

rep invariant as a design tool
  • how a strong invariant helps in clarity and performance

structural induction
  • what makes rep invariant reasoning modular

example
  • representing sets as lists
introduction
limitations of testing

doesn’t find all bugs

• systematic testing is hard: easy to miss cases
• spec partitions don’t cover code
  eg, testing set ops, might ignore order in which sets are constructed

can’t assess level of confidence

• when no more bugs found
• so when do you ship?
• what risks can you afford to take using software?

“Program testing can be used to show the presence of bugs, but never to show their absence!”
E.W. Dijkstra, Structured programming (EWD268)
http://www.cs.utexas.edu/users/EWD/
more powerful testing

two main strategies for making testing more powerful

stronger coverage

‣ require that all test predicates evaluate to true/false in all combinations
‣ a variant of this, called “MCDC” is used by the FAA for aviation software
‣ very expensive and burdensome
‣ big impact, but mainly because it forces good requirements

automatic generation

‣ generate huge suite of cases fully automatically
‣ sometimes random generation (“fuzz testing”) is effective
‣ many research projects on this

but fundamentally, these don’t address the confidence problem
so why bother?

testing is very expensive
  • typically 30% of entire project cost

but not testing is even more expensive
  • 2002 NIST study said bugs cost US $60B/year, 36% to developers
    http://www.computerworld.com/managementtopics/management/itspending/story/0,10801,72245,00.html
  • risks to civic infrastructure: voting, air traffic, energy, etc
  • 200,000 pacemakers recalled last decade
    http://www7.nationalacademies.org/cstb/pub_dependable.html
an alternative to testing

informal reasoning
  • reason that code satisfies spec by informal argument
  • programmer argues to another person (not herself!) that code works
  • called “code review”; found to be extremely effective

formal reasoning
  • mathematical proof that code meets spec
  • can be checked fully automatically
  • construction typically not fully automatic, but major research progress
our view

reasoning or testing?

- techniques complement each other
- testing essential for end-to-end behaviour
- reasoning essential for quality, and often less work

formal or informal?

- formal proof beyond this course
- but big advantage from using formal concepts in informal arguments
- ie, rep invariants, abstraction functions, pre/post
review of key concepts
pre/post

general form of spec: pre and post conditions

\[ T \text{ m } (T_1 \text{ a}_1, T_2 \text{ a}_2, \ldots) \]

**requires** some constraint involving args this, a1, a2, ...

**returns** some constraint involving result and args

precondition

‣ is a disclaimer, not a condition to be checked

‣ obligates caller, assumed by implementor

postcondition

‣ obligates implementor, assumed by caller

modularity

‣ method correctness is defined **independently** of other methods
**rep invariant R**

- defines set of legal representation values

**abstraction function A**

- interprets legal representation values as abstract values
proof obligation

suppose we have

- precondition \( \text{pre}: \text{Abstract} \rightarrow \text{Bool} \)
- postcondition \( \text{post}: \text{Abstract, Abstract} \rightarrow \text{Bool} \)
- rep invariant \( \text{R}: \text{Concrete} \rightarrow \text{Bool} \)
- abstraction function \( \text{A}: \text{Concrete} \rightarrow \text{Abstract} \)

then method is correct if

- this formula is valid
  \[
  \forall c, c': \text{Concrete} \mid \text{pre}(A(c)) \land R(c) \implies \text{post}(A(c), A(c')) \land R(c')
  \]

why does this work?

- and why not just this?
  \[
  \forall c, c': \text{Concrete} \mid \text{pre}(A(c)) \implies \text{post}(A(c), A(c'))
  \]
rep invariant as a design tool
how rep invariants help

strong rep invariant

- means implementor an assume more on entry
- can make it easier to write code (can ignore some cases)
- and allows performance tricks

example: set implemented as list. which of these R is useful and why?

```java
class ListSet implements Set {
    List elements;
    ...
}

- R(l) = l.elements != null
- R(l) = Sorted (l.elements)
- R(l) = l.elements != null => l.elements.size() > 0
- R(l) = all i, j | l.elements[i].equals (l.elements[j]) => i = j
```
structural induction
inductive generators

last time

- identify **generators** $\subseteq$ constructors $\cup$ producers, sufficient to make all values

examples

- naturals generated by $\{\text{zero}, \text{incr}\}$
  $$\text{Nat} = \text{incr}(\text{Nat}) \cup \text{zero}()$$

- lists generated by $\{\text{emptyList}, \text{add}\}$
  $$\text{List}<E> = \text{add}(E, \text{List}<E>) \cup \text{emptyList}()$$

- sets generated by $\{\text{emptySet}, \text{add}\}$
  $$\text{Set}<E> = \text{add}(E, \text{Set}<E>) \cup \text{emptySet}()$$
structural induction

suppose we want to show that

- a property $I$ holds for all values of a set $S$
- and $S$ is generated by $c: \to S$ and $p: S \to S$

can prove inductively

- base case: $I(c)$ holds
- inductive case: if $I(s)$ holds, then $I(p(s))$ holds

data example: $\forall i: \text{Nat} \mid 2^i \geq 1$
- recall $\text{Nat} = \text{incr(\text{Nat})} \cup \text{zero()}$
- base case: $I(\text{zero()}) = 2^0 = 1$
- inductive case: $2^{i+1} = 2^i \times 2$ so, if $I(i)$ then $I(\text{incr}(i))$
rep invariants

want to show \( R(c) \) for all reachable concrete values \( c \)

\( \cdot \) \( R \) is a property that can be proved by structural induction

\( \cdot \) need to consider all constructors and producers (why?)

proving a rep invariant

\( \cdot \) base case: show that each constructor \( C \) establishes \( R \)

\hspace{1cm} \text{returns } c \text{ such that } R(c)

\( \cdot \) induction case: show that producer maintains \( R \)

\hspace{1cm} \text{if passed } c \text{ such that } R(c), \text{ returns } c' \text{ such that } R(c')
example: sets as lists
list specification

public interface List<E> {
    // producers
    List<E> add (E e);
    List<E> remove(E e);

    // observers
    E first();
    List<E> rest();
    boolean contains (E e);
    int size ();

    boolean equals(Object o);
    int hashCode();
}
public class NonEmptyList<E> implements List<E> {
    private E element;
    private List<E> rest;
    private int size;

    private NonEmptyList(E e, List<E> r) {
        element = e;
        rest = r;
        size = r.size() + 1;
    }

    public NonEmptyList(E e) {
        element = e;
        rest = new EmptyList<E>();
        size = 1;
    }

    public List<E> add(E e) {
        return new NonEmptyList<E> (e, this);
    }

    public E first() {
        return element;
    }

    public int size () {
        return size;
    }
    ...

exercises
• find a rep invariant involving size
• how does it help?
• how is it preserved?
public class EmptyList<E> implements List<E> {

    public EmptyList () {
    }

    public List<E> add(E e) {
        return new NonEmptyList<E> (e);
    }

    public List<E> remove(E e) {
        return this;
    }

    public E first() {
        assert false : "List.rest(null)";
        return null;
    }

    ...

    public boolean contains (E e) {
        return false;
    }

    public int size () {
        return 0;
    }

    ...
}
public interface Set<E> {
    // producers
    Set<E> add (E e);
    Set<E> remove (E e);
    Set<E> addAll (Set<E> s);
    Set<E> removeAll (Set<E> s);

    // observers
    E choose ();
    boolean contains (E e);
    boolean isEmpty ();
    int size ();

    // observers from Object
    boolean equals(Object o);
    int hashCode();
}
public class ListSet<E> implements Set<E> {
    List<E> elements;

    private ListSet (List<E> es) {
        assert es != null;
        elements = es;
    }

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

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

    public Set<E> remove (E e) {
        if (isEmpty()) return this;
        E x = elements.first();
        Set<E> s = new ListSet<E> (elements.rest());
        if (x.equals(e))
            return s;
        else {
            return s.remove(e).add(x);
        }
    }
...
The easiest way to preserve R

Here’s the code of a method of ListSet:

why is checking R easy in this case?

```java
public Set<E> addAll (Set<E> s) {
    if (s.isEmpty()) return this;
    else {
        E e = s.choose ();
        Set<E> s2 = s.remove(e);
        return add(e).addAll(s2);
    }
}
```
reasoning about a method

given rep invariant and abstraction function

\[ R(s) = s.\text{elements} \neq \text{null} \&\& \text{no_dups}(s.\text{elements}) \]
\[ A(s) = \{ e \mid \text{exists } i \mid s.\text{elements}[i] = e \} \]

argue that this method

\- preserves the rep invariant
\- satisfies the spec

```java
/**
 * Add element to set
 * @return this U {e}
 */
public Set<E> add (E e) {
    if (elements.contains (e)) return this;
    else return new ListSet<E> (elements.add (e));
}
```
argue correctness of this method
• how is rep invariant exploited?

/**
 * Remove element from set
 * @return this \ {e}
 */
public Set<E> remove (E e) {
   if (isEmpty()) return this;
   E x = elements.first();
   Set<E> s = new ListSet<E> (elements.rest());
   // A(this) = s U \{x\} and x not in s
   if (x.equals(e))
      // x = e => A(this) \ {e} = s
      return s;
   else {
      // x != e => A(this) \ {e} = (s \ {e}) U \{x\}
      return s.remove(e).add(x);
   }
}
a coding challenge

take this implementation of removeAll

```java
/**
 * Intersection
 * @requires
 * @return this \ s
 */
public Set<E> removeAll (Set<E> s) {
    if (isEmpty()) return this;
    else {
        E e = elements.first();
        Set<E> s2 = new ListSet<E>(elements.rest()).removeAll(s);
        if (s.contains(e)) return s2;
        else return s2.add(e);
    }
}
```

and

- modify it so that it exploits the rep invariant
puzzles for the reader

look at the implementation of toString in the repository

‣ what rep invariant ensures that it terminates?
‣ how is this invariant preserved?
summary

reasoning
• more powerful than testing: covers all cases
• modular: each method considered independently
• but still error prone, so testing is essential

rep invariant
• essential tool for designing abstract data types
• strong invariant improves code clarity and performance

structural induction
• key to ensuring that R holds on all reachable values