Representation Invariants
6.005 Fall 2010

1. Introduction

In this lecture, we describe two tools for understanding abstract data types: the representation invariant and the abstraction function. The representation invariant describes whether an instance of a type is well formed; the abstraction function tells us how to interpret it. Representation invariants can amplify the power of testing. It's impossible to code an abstract type or modify it without understanding the abstraction function at least informally. Writing it down is useful, especially for maintainers, and crucial in tricky cases.

2. What is a Rep Invariant?

A representation invariant, or rep invariant for short, is a constraint that characterizes whether an instance of an abstract data type is well formed, from a representation point of view. Mathematically, it is a formula over the representation of an instance; you can view it as a function that takes objects of the abstract type and returns true or false depending on whether they are well formed:

$$RI : Object \rightarrow Boolean$$

Consider the Java LinkedList class which has a field, header, that holds a reference to an object of the class Entry. This object has three fields: element, which holds a reference to an element of the list; prev, which points to the previous entry in the list; and next, which points to the next element.

To see how this works, let's look at some sample operations of our LinkedList class. The representation is declared in Java like this:

```java
class LinkedList {
    Entry header;
    int size;
    class Entry {
        Object element;
        Entry prev;
        Entry next;
        Entry (Object e, Entry p, Entry n) {
            element = e; prev = p; next = n;
        }
    }
    . . .
}
```
Here is an example of a linked list with two elements of the Java implementation java.util.LinkedList.

The representation invariant is a constraint that holds for every instance of the type. Let us state some properties of the linked list objects by looking at the example above.

- The header field holds a reference to an object of class Entry. This property is important but not very interesting, since the field is declared to have that type; this kind of property is more interesting for the contents of polymorphic containers such as vectors, whose element type cannot be expressed in the source code.
- The header field cannot be null.
- Each of the next and prev arrows point to exactly one entry.
- Each entry is pointed to by exactly one other entry's next field, and by exactly one other entry's prev field.
- Each Entry points to at most one Object.

There are other properties of the representation invariant.

- When there are two $e_1$ and $e_2$ entries in the list, if $e_1$.next = $e_2$, then $e_2$.prev = $e_1$.
- The dummy entry at the front of the list has a null element field.

There are also properties that do not appear in the diagram. The representation of LinkedList has a field size that holds the size of the list. A property of the representation invariant is that size is equal to the number of entries in the list representation, minus one (since the first entry is a dummy).

When examining a representation invariant, it is important to notice not only what constraints are present, but also which are missing. In this case, there is no requirement that the element field be non-null, nor that elements not be shared. This is what we’d expect: it allows a list to contain null references, and to contain the same object in multiple positions.
Let's summarize our rep invariant informally:

- for every instance of the class LinkedList
  - the header field is non-null
  - the header field has a null element field
  - there are \((\text{size} + 1)\) entries
  - the entries form a cycle starting and ending with the header entry
  - for any entry, taking prev and then next returns you to the entry, and vice versa

We can also write this a bit more formally:

\[
\text{all } p: \text{LinkedList} \mid
p.\text{header} \neq \text{null} \&\&
\text{p.header.element} = \text{null} \&\&
\text{p.size} + 1 = | p.\text{header.}*\text{next} | \&\&
p.\text{header} = p.\text{header.}*\text{next} \text{p.size} + 1 \&\&
\text{all } e \text{ in } p.\text{header.}*\text{next} | e.\text{prev.next} = e \&\&
\text{all } e \text{ in } p.\text{header.}*\text{next} | e.\text{next.prev} = e
\]

To understand this formula, you need to know that

- for any expression \(e\) denoting some set of objects, and any field \(f\), \(e.\*f\) denotes the set of objects you get if you follow \(f\) from each of the objects in \(e\);
- \(e.\*f\) means that you collect the set of objects obtained by following \(f\) any number of times from each of the objects in \(e\);
- \(| e |\) is the number of objects in the set denoted by \(e\).

So \(p.\text{header.}*\text{next}\) for example denotes the set of all entries in the list, because you get it by taking the list \(p\), following the \(\text{header}\) field, and then following the \(\text{next}\) field any number of times.

One thing that this formula makes very clear is that the representation invariant is about a single linked list \(p\). Another fine way to write the invariant is this:

\[
\text{R}(\text{p}) = p.\text{header} \neq \text{null} \&\&
\text{p.header.element} = \text{null} \&\&
\text{p.size} + 1 = | p.\text{header.}*\text{next} | \&\&
p.\text{header} = p.\text{header.}*\text{next} \text{p.size} + 1 \&\&
\text{all } e \text{ in } p.\text{header.}*\text{next} | e.\text{prev.next} = e \&\&
\text{all } e \text{ in } p.\text{header.}*\text{next} | e.\text{next.prev} = e
\]

in which we view the invariant as a boolean function. This is the point of view we'll take when we convert the invariant to code as a runtime assertion.

**Question:** Are all of the clauses in \(\text{R}(\text{p})\) above necessary? Can you prove that some imply the others?

The choice of invariant can have a major effect both on how easy it is to code the implementation of the abstract type, and how well it performs. Suppose we strengthen our invariant by requiring that the \(\text{element}\) field of all entries other than the header is non-null. This would allow us to detect the header entry by comparing its element to null; with the current invariant, operations that require traversal of the list must count entries instead or compare to the header field. Suppose, conversely, that we weaken the
invariant on the next and prev pointers and allow prev at the start and next at the end to have any values. This will result in a need for special treatment for the entries at the start and end, resulting in less uniform code. Requiring prev at the start and next at the end both to be null doesn’t help much.

3. Inductive Reasoning

The rep invariant makes modular reasoning possible. To check whether an operation is implemented correctly, we don’t need to look at any other methods. Instead, we appeal to the principle of induction. We ensure that every constructor creates an object that satisfies the invariant, and that every mutator and producer preserves the invariant: that is, if given an object that satisfies it, it produces one that also satisfies it. Now we can argue that every object of the type satisfies the rep invariant, since it must have been produced by a constructor and some sequence of mutator or producer applications.

To see how this works, let’s look at some sample operations of our LinkedList class. Here’s our constructor:

```java
public LinkedList () {
    size = 0;
    header = new Entry (null, null, null);
    header.prev = header.next = header;
}
```

Notice that it establishes the invariant: it creates the dummy element, forms the cycle, and sets the size appropriately.

The mutator add takes an element and adds it to the end of the list:

```java
public void add (Object o) {
    Entry e = new Entry (o, header.prev, header);
    e.prev.next = e;
    e.next.prev = e;
    size++;
}
```

To check this method, we can assume that the invariant holds on entry. Our task is to show that it also holds on exit. The effect of the code is to splice in a new entry just before the header entry, i.e., this new entry becomes the last entry in the next chain, so we can see that the constraint that the entries form a cycle is preserved. Note that one consequence of being able to assume the invariant on entry is that we don’t need to do null reference checks: we can assume that e.prev and e.next are non-null, for example, because they are entries that existed in the list on entry to the method, and the rep invariant tells us that all entries have non-null prev and next fields.

Finally, let’s look at an observer. The operation getLast returns the last element of the list or throws an exception if the list is empty:

```java
public Object getLast () {
    Object p = header.prev;
    if (p == header) throw new NoSuchElementException ();
    return p.element;
}
```

Note first of all the unprotected references. We don’t need to check that header is non-null before derefencing its prev field, and we don’t need to check that header.prev is
non-null either. The rep invariant tells us that these will never be null, even if the list is empty. We don’t need to search along the list until we get to the end, because the rep invariant tells us that the entries form a cycle. A poor design, without a good rep invariant, might make this simple method quite complicated. Checking that the invariant is preserved is trivial in this case, since there are no modifications.

Let’s say that we can prove that the rep invariant holds for each constructed object. Further assume that we can prove that the rep invariant holds for every object obtained as a result of a mutator or producer, provided the invariant holds for all objects given to the mutator or producer. Is there any reason to check the invariant as a runtime assertion in the code? Yes, because rep exposure may result in ill-formed objects, when code outside of the type’s methods, modifies objects. Thus, even if you are completely confident in your inductive reasoning, it still makes sense to check the invariant at the beginning of each mutator or producer.

Note: Observer methods are allowed to mutate the rep as long as they do not change the abstract value represented (see next lecture). For this reason, we recommend that observer methods be treated in a similar manner to mutators.

4. Interpreting the Representation

Consider the mutator add again, which takes an element and adds it to the end of the list:

```java
public void add (Object o) {
    Entry e = new Entry (o, header.prev, header);
    e.prev.next = e;
    e.next.prev = e;
    size++;
}
```

We checked that this operation preserved the rep invariant, by correctly splicing a new entry into the list. What we didn’t check, however, was that it was spliced into the right position. Is the new element inserted into the start or the end of the list? It looks as if it’s at the end, but that assumes that the order of entries corresponds to the order of elements. It would be quite possible (although perhaps a bit perverse) for a list \( p \) with elements \( o_1, o_2, o_3 \) to have:

- \( p.\text{header}.\text{next}.\text{element} = o_3 \)
- \( p.\text{header}.\text{next}.\text{next}.\text{element} = o_2 \)
- \( p.\text{header}.\text{next}.\text{element} = o_1 \)

To resolve this problem, we need to know how the representation is interpreted: that is, how to view an instance of LinkedList as an abstract sequence of elements. This is what the abstraction function provides. The abstraction function for our implementation is:

```java
A(p) =
    if p.\text{header}.\text{next} = p.\text{header} then
        the empty sequence
    else
        the sequence
            <\text{p.header.next.element}, p.\text{header}.\text{next}.\text{next}.\text{element}, ..
        ending with the first entry e such that e.next = p.header
```

\[5\]
5. An Example of Rep Exposure Violating a Rep Invariant

Representation exposure occurs when mutable objects are passed in to build an underlying representation.

Consider the following code where HashSet can be viewed as a hash table without values. The representation invariant of a hash table is that keys are in the slots determined by their hash codes.

1. Set<List> s = new HashSet<List>();
2. List<Object> x = new LinkedList<Object>();
3. s.add(x);
4. x.add(new Object());

x was made part of the representation in Line 3. Line 4 mutated the underlying representation.

x has been modified in Line 4, and now s.contains(x) will likely return false. If you think that this is acceptable, consider that there is no x for which s.contains(x) will consistently return true, even though s.size() returns 1!

The representation invariant of the hash table that keys are placed in slots according to their hash codes is broken when the keys are modified. This is why the Java API for hash set says: **Great care must be exercised if mutable objects are used as set elements. The behavior of a set is not specified if the value of an object is changed in a manner that affects equals comparisons while the object is an element in the set. A special case of this prohibition is that it is not permissible for the set to contain itself as an element.**

6. Another Example of a Rep Invariant: Heaps

A heap is an array object that is viewed as a nearly complete binary tree.

![Binary Tree Image]

Root is A[1].

For Node with index i
\[
PARENT(i) = \text{floor}(i / 2) \\
LEFT(i) = 2i \\
RIGHT(i) = 2i + 1 \\
\]

**Note:** NO POINTERS!

length[A]: number of elements in the array
heap-size[A]: number of elements in the heap stored within array A

(Partial) Rep Invariant:

\[
\text{heap-size}[A] \leq \text{length}[A] \\
\text{Max-Heap Property: For every node } i \text{ other than the root } A[PARENT(i)] \geq A[i] \\
\]

Heap operations insert, find_max, extract_max, etc. We need to check the rep invariant before and after each operation, or prove that the invariant is maintained by the method and that rep exposure has not occurred.

### 7. Summary

Why use rep invariants? Recording the invariant can actually save work:
- It makes modular reasoning possible. Without the rep invariant documented, you might have to read all the methods to understand what’s going on before you can confidently add a new method.
- It helps catch errors. By implementing the invariant as a runtime assertion, you can find bugs that are hard to track down by other means.