ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Exercise 13

Invariants

December 20, 2013

Task 1

A technique to represent a complete binary tree `T` using an array `A`, is:

- store the root in `A[0]`
- for any node `N` stored in `A[i]`, store the children of `N` to `A[2i+1]` and `A[2i+2]`.

The size of the array should be equal to $2^{h+1} - 1$, where h is the height of the tree.

Consider the following invariant on a complete binary tree of integers: *any non-leaf node stores the sum of the integers stored in its two children.* Let us call this invariant **U** (for "undented"; cf. "dented invariants" on Lecture 9, Slide 11).

The following class uses the above-mentioned representation.

```
final class CompleteBinaryTree
{
   private int[] theTree;

   public CompleteBinaryTree(int h)
   {
      theTree = new int[Math.pow(2,h+1)-1];
      for(int i=0; i<theTree.length; i++)
         theTree[i]=0;
   }

   /// requires 0 ≤ i < theTree.length
   public int getNode(int i) { return theTree[i]; }

   /// requires theTree.length/2 ≤ i < theTree.length
   // this means i must be a leaf
   public void addToLeaf(int i, int s)
   { addToNode (i, s); }

   private void addToNode(int i, int s)
   {
      theTree[i]+=s;
      if (i>0) addToNode((i-1)/2, s);
   }
}
```

(a) Write formally the invariant **U**.

(b) The method `addToNode` does not preserve **U**. Instead, its purpose is to fix **U**, when it is temporarily broken. Describe how this is done.

(c) Describe informally the precondition under which the method `addtoNode` has to be called, such that **U** holds when the method terminates.

(d) Dent **U** accordingly so that the precondition above is formally expressible. Hint: Denting usually uses a single boolean field (see Lecture 9, Slide 11). Here, you need more than one boolean field.

(e) Add assignments to the new boolean fields in the bodies of all the methods and write specifications for all the methods. All methods must preserve the dented invariant.

(f) Explain why the public interface of the class preserves **U**.

Solution

(a) The invariant of the class, apart from **U**, should also contain the following conjuncts:

`theTree≠null ∧ ∃h:int. h≥0 ∧ theTree.length=2^(h+1)-1`

This part of the invariant is assumed throughout the solution, and we will not refer to it again.

The invariant **U** can be written as follows:

```
∀i.0≤i<theTree.length/2⇒
      theTree[i] = theTree[2*i+1]+theTree[2*i+2]
```

Note that the condition `0≤i<theTree.length/2` says that node `i` is not a leaf. Note also that "height" means the maximum distance of the root to the leaves (so a single node is a 0-height tree)

(b) When `addToLeaf` is called on a leaf, a sequence of recursive calls to `addToNode` begins. The first call adds a number `s` to the leaf, which temporarily breaks the invariant, because the parent of that leaf no longer holds the correct sum. Each subsequent call of `addToNode` corrects the sum of its current node, similarly making the sum of its parent (if there is one) outdated. The calls to `addToNode` happen recursively all the way up from the leaf to the root, at which point the invariant is fixed.

(c) The precondition is as follows: either (i) the method `addToNode` is called on a leaf or (ii) the invariant must be broken exactly at the node on which we call `addToNode`. In the latter case, the sum of the children of that node must be exactly `s` less than what it is supposed to be.

(d) We can dent the invariant in the following way: Introduce a boolean array `b`. For every non-leaf `i`, the flag `b[i]` is true if and only if the **U** has to hold at node `i`. More formally, the dented version of the invariant is:

```
∀i.0≤i<theTree.length/2 ∧ b[i]⇒
      theTree[i] = theTree[2*i+1]+theTree[2*i+2]
```

where the field `b` is declared as `bool[] b;`

This denting allows us to break **U** at any node in the tree, which makes the precondition described in (c) easily expressible.

Remember that the invariant must also specify that `b` is not null, and that the size of `b` is equal to the number of non-leaf nodes in the tree.

(e) Here is the code together with the new field:

```
final class CompleteBinaryTree
{
   bool[] b;
   private int[] theTree;

   /// invariant theTree≠null ∧ b≠null
```

```
    /// invariant ∃h:int. h≥0 ∧ theTree.length=2^(h+1)-1 ∧ b.length=2^h-1
    /// invariant: as mentioned in (d)

    public CompleteBinaryTree(int h)
       /// ensures ∀i.b[i]
    {
       theTree = new int[Math.pow(2,h+1)-1];
       for(int i=0; i<theTree.length; i++)
          theTree[i]=0;
       b = new bool[Math.pow(2,h)-1];
       for(int i=0; i<b.length; i++) b[i]=true;
    }

    public void addToLeaf(int i, int s)
       /// requires theTree.length/2 ≤ i < theTree.length
       /// requires ∀j.b[j]
       /// ensures theTree[i]=old(theTree[i+1])+s
       /// ensures ∀j.b[j]
    { addToNode (i, s); }

    private void addToNode(int i, int s)
       /// requires 0 ≤ i < theTree.length
       /// requires i<theTree.length/2⇒
         ¬b[i] ∧ theTree[i]=theTree[2*i+1]+theTree[2*i+2]-s
       /// requires ∀j. i≠j⇒b[j]
       /// ensures theTree[i]=old(theTree[i+1])+s
       /// ensures ∀j.b[j]
    {
       theTree[i]+=s;
       if(i<b.length) b[i]=true;
       if (i>0)
       {
          b[(i-1)/2]=false;
          addToNode((i-1)/2, s);
       }
    }
 }
```

(f) The claim is as follows: all methods preserve the dented invariant, and the public methods preserve the condition `∀j.b[j]`, which guarantees the undented invariant **U**.

Task 2

Consider the following example

```
class Redundant {
  int a, b;
  Logger l;

  ///invariant a == b

  public setLogger(Logger l) { this.l = l; }

  public void set( int v ) {
    a = v;
    l.log( "Inside set" );
    b = v;
  }

  public int div( int v ) {
    return v / ( a - b + 1 );
  }
```

```
}

class Logger {
  private Redundant r;

  public Logger(Redundant r) { this.r = r; }
  public void log( String m ) {
    System.out.println( m + r.div( 5 ) );
  }
}
```

- Write client code that causes `div` to throw an exception
- Suppose that we change the implementation of `set` as follows:

```
public void set( int v ) {
  a = v;
  b = v;
}
```

  Can we still cause `div` to throw an exception by writing code outside the two classes?

Solution

- Here is an example (note that Java initializes integers to 0):

```
Redundant r = new Redundant();
Logger l = new Logger(r);
r.setLogger(l);
r.set(-1);
```

- We can break the code again. First we override `set` to re-introduce the unsafe callback in the subclass:

```
class ReallyRedundant extends Redundant
{
  public void set( int v ) {
    a = v;
    l.log( "Inside set" );
    b = v;
  }
}
```

  And then we use the same trick again:

```
Redundant r = new ReallyRedundant();
Logger l = new Logger(r);
r.setLogger(l);
r.set(-1);
```

Task 3

Consider the following Java classes:

```
class Vector {
   public int x, y;
   Vector(int x, int y) {
      this.x=x;
      this.y=y;
   }
}

class SumVectors {
   public Vector[] a=new Vector[0];
```

```
   public void insert(Vector vct) {
      Vector[] o=a;
      a=new Vector[a.length+1];
      for(int i=0; i<o.length; i++) a[i]=o[i];
      a[a.length-1]=vct;
   }

   public Vector sum() {
      int x=0, y=0;
      for(Vector v : a) {
         x+=v.x;
         y+=v.y;
      }
      return new Vector(x, y);
   }
}
```

- Annotate the classes with specifications that ensure that there is no null-pointer dereferencing, that method insert inserts a new `Vector` object in the end of the array `a`, and that method `sum` computes the sum of all vectors in the array `a`.
- Annotate the following class with invariants, such that it is a behavioural subtype of `SumVectors`:

```
class FastSumVectors extends SumVectors
{
   int sx=0, sy=0;

   public void insert(Vector vct)    {
      super.insert(vct);
      sx+=vct.x; sy+=vct.y;
   }

   public Vector sum() {
      return new Vector(sx, sy);
   }
}
```

Solution

- We need the following specifications:

```
class SumVectors
{
   public Vector[] a=new Vector[0];

   /// invariant a≠null ∧ ∀ v:a. v≠null

   public void insert(Vector vct)
   /// requires vct≠null
   /// ensures a.length=old(a).length+1
   /// ensures ∀i:int. 0≤i<old(a).length ⇒
   ///         a[i]=old(a)[i]
   /// ensures a[old(a).length]=vct
   { ... }

   public Vector sum()
   /// ensures result.x = Σ_{i=0}^{a.length−1} a[i].x
   /// ensures result.y = Σ_{i=0}^{a.length−1} a[i].y
   { ... }
}
```

Note that there are some additional annotations that the questions doesn't ask for, which could nevertheless be useful:

  - We might want to ensure that the element added to the array by `insert()` is indeed identical to the argument that was passed in (i.e. `vct` does not change).
  - We could annotate `sum()` to indicate that it is pure.

- We only need one invariant:

  $\texttt{sx}=\Sigma_{i=0}^{a.length-1}\texttt{a[i].x} \wedge \Sigma_{i=0}^{a.length-1}\texttt{a[i].y}$