Exercise 3

Behavioural Subtyping

October 15th, 2010

In-class Assessment: One or more questions from this sheet will be used for the in-class assessment. No notes are allowed during the assessment.

1. Let `SortedArray` be a Java class, which supports a single private field `A`. The field `A` must be a sorted (in increasing order) array of integers with no duplicates. The following is a method for the insertion of a value into the array:

```
void insert (int x)
{
      int[] B = new int[A.length+1];
      int i = 0;
      while (i<A.length && A[i]<x)
      {
            B[i]=A[i];
            i++;
      }
      B[i]=x;
      while (i<A.length)
      {
            B[i+1]=A[i];
            i++;
      }
      A=B;
}
```

   Write an appropriate invariant for the class, as well as a pre- and postcondition for the method `insert`. Make the postcondition as strong as possible. In your answers, you may use the logical quantifiers ∀ and ∃.

2. Let `C` be a class with an integer field `x` and a method `m`. Let `m` have
   (a) Precondition `x>0`
   (b) Postcondition `x<1`

   Suppose now that there is a class `D` with an integer field `x` and a method `m`. In which of the following cases does the specification of `m` in `D` permit `D` to be a behavioural subtype of `C`?

   - Pre `x>0` Post `x<-1`
   - Pre `x>0` Post `x<2`
   - Pre `x>-1` Post `x<1`
   - Pre `x>2` Post `x<1`
   - Pre `x>-4` Post `x<-old(x)*old(x)`
   - Pre `true` Post `false`

3. Consider the example of behavioural subtyping in Slide 59.

```
class Super
{
      // requires p == p*p
      // ensures p < result
      int foo( int p ) { … }
};

class Sub extends Super
{
      // requires p == 0 || p == 1
      // ensures result == 2
      int foo( int p ) { … }
}
```

Suppose that we try to prove this behavioural subtyping. According to our requirements, the precondition of `foo` in the superclass should be stronger than that in the subclass:

`p==p*p ==> p==0 || p==1` (1)

and its postcondition in the subclass should be stronger than that in the superclass:

`result==2 ==> p<result` (2)

Formula (1) is a theorem, but Formula (2) is not!

What is wrong here? Is `Sub` a behavioural subtype of `Super`?

If not, then exhibit an example of an implementation of `foo` in `Sub` that violates the contract of `Super`.

If yes, then formulas (1) and (2) are too strong to prove the behavioural subtyping. Propose weaker rules for the proof of behavioural subtyping, that circumvent this problem.

4. You are provided with three classes `ArrayNonDecreasing`, `ArrayIncreasing`, and `ArrayNoDuplicates`. Each of them has a private field `content` of type array of integers. The classes have the following properties:
   (a) `ArrayNonDecreasing` – elements of `content` must be in non-decreasing order
   (b) `ArrayIncreasing` - elements of `content` must be in increasing order
   (c) `ArrayNoDuplicates` – elements of `content` must be unique

   - Write invariants that express these properties
   - Suppose that there are no mutator methods (i.e. methods that change the state). Could there be any behavioural subtype relations between these classes? If yes, then describe them.
   - Can you find a mutator method that, when added to each of the classes, would make impossible the behavioural subtyping relations identified in the previous question? You should consider appropriate pre- and postconditions for your chosen method.

5. Suppose that we have a database, for which we want an "automated key generation" feature. This means that each time the user inserts a new tuple, a unique key is automatically generated for the tuple by the system. An obvious way to do that is to write a counter, which increments by 1 the value that it returns each time it is called. The method that generates a new key is called `generate`.

   (a) Write a Java class `IncCounter` and an accompanying specification for such a counter.

   (b) Annotate the following Java class with specifications and show that it is not a behavioural subtype of `IncCounter`.

```
class DecCounter
{
  int key;
  DecCounter () { key = 0; }
  int generate () { return key--; }
}
```

   (c) **(Harder)** Write an abstract class `GenerateUniqueKey` together with a specification, such that both `IncCounter` and `DecCounter` are behavioural subtypes of `GenerateUniqueKey`. In the specification, you may use helper methods and fields.