ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Exercise 4

Contracts and Behavioral Subtyping

October 21, 2022

Task 1

Let `SortedArray` be a Java class which has a field `A` of type `int[]`. We say that a `SortedArray` object is in a *valid* state if the field `A` is not `null` and stores a sorted array (in increasing order) with no duplicates. The method `insert` adds the value of `x` into the array:

```
class SortedArray {
  int[] A;

  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;
  }
}
```

Give an appropriate invariant for the class, as well as a precondition and a postcondition for the method `insert`, such that only valid states can be reached. In particular, the precondition should be as permissive as possible, and the postcondition should precisely describe the behaviour of the method. You may use quantifiers $(\forall, \exists)$ in your annotations. Note that the invariant is always automatically added to the precondition and postcondition, so there is no need to repeat it.

Hint: Consider what happens when the item to be inserted into the array already exists. Do *not* change the implementation to avoid this situation.

solution

```
class sortedArray{
   int[] A;
   /// invariant A ≠ null
   /// invariant ∀ i:int | 0 ≤ i ∧ i < A.length-1 ⇒ A[i] < A[i+1]

   /// requires  ∀ i:int | 0 ≤ i ∧ i < A.length ⇒ x ≠ A[i]
   /// ensures A.length = old(A.length) + 1
   /// ensures
   /// ∃ j:int |
   ///    (0 ≤ j ∧ j < A.length)
```

```
    ///    ∧ A[j] = x
    ///    ∧ ( ∀ i:int | (0 ≤ i ∧ i < j ⇒ A[i] = old(A[i]))
    ///    ∧ ( ∀ i:int | (j < i ∧ i < A.length ⇒ A[i] = old(A[i-1]))
    void insert (int x){...}
}
```

Another approach to express the specification of `insert` is as follows: first, we introduce an auxiliary function `contains`:

```
contains(L, x) = ∃ j:int | (0 ≤ j ∧ j < L.length ∧ L[j] = x)
```

Using `contains` we can express the specifications of `insert` as follows:

```
requires ¬contains(A, x)
ensures ∀ y:int | contains(A, y) ⇔ (y = x ∨ contains(old(A), y))
```

Task 2

Alice and Bob are two software developers. Alice is writing a small class `Cell` that stores an integer. The class supports methods for setting/getting/increasing the integer. Bob is going to write code that uses the class `Cell`.

Here are the contracts of the methods (the bodies are omitted):

```
class Cell {

  public int n;
  // this field is public for simplicity;
  // generally this is not a recommended practice

  /// requires true
  /// ensures n == p
  public void set(int p) { ... }


  /// requires true
  /// ensures result == n && n == old(n)
  public int get() { ... }

  /// requires true
  /// ensures n > old(n)
  public void inc() { ... }
}
```

In this exercise we will experiment with modifying the specifications. In particular, if we modify a specification, it might become:

- *more restrictive* for a party. For example, a specification that is more restrictive for Alice does not allow some implementations that were OK with the old specification. A specification that is more restrictive for Bob might mean that a piece of code that Bob wrote cannot guarantee something that it had guaranteed before.
- *more flexible* for a party. A specification S is more flexible than a specification S' for a party P if S' is more restrictive than S for P.
- it might be the case that the new specification is neither more restrictive nor more flexible for a party. For example, the new specification makes some previously correct code illegal, while it also makes some previously illegal code correct.

For example, if we modify the postcondition of `get` such that:

```
result == n || result == -n
```

the specification becomes more flexible for Alice, because she is allowed the, previously illegal, implementation of `get`:

```
return n > 5 ? n : -n;
```

while, at the same time, it becomes more restrictive for Bob, because the following code

```
c.set(3); x = c.get();
```

does not guarantee the postcondition `x == 3` anymore.

For each of the following specification changes (subtasks a-d), do the following:

(i) Write formally the new pre/postconditions (not invariants). Only write the pre/postconditions that change.

(ii) Compare the flexibility of the new specifications to the old ones, from the point of view of both Alice and Bob.

(iii) Justify your answers for both parties by *providing code*.

Note that a postcondition should be satisfiable for any valid pre-state. You can assume that the implementations of the methods do not call each other and that there are no integer overflows.

(a) It is only allowed to set `n` to a strictly positive value.

solution

This amounts to adding the precondition `p > 0` to `set`. This specification is more flexible for Alice; for example the following, previously incorrect, implementation is now valid:

```
if(p > 0) n = p;
```

On the other hand, this is more restrictive for Bob, because the code

```
c.set(-1);
```

is not allowed by the new precondition anymore.

(b) `inc` should increase `n` by exactly one.

solution

This changes the postcondition of `inc` to `n == old(n) + 1`. Alice is more restricted, since she cannot do this anymore:

```
n = n + 2;
```

Bob is more flexible. Now

```
c.set(4); c.inc(); x = c.get();
```

guarantees the postcondition `x == 5`, which it didn't before.

(c) `inc` should increase `n` by a positive non-zero amount, but it should guarantee that the final value of `n` is strictly positive.

solution

This conjoins the postcondition `n > 0` to `inc`. The implementation from (b) still does not work for Alice, who is more restricted. Bob, on the other hand, is more flexible:

`c.inc(); x = c.get();`

guarantees the postcondition `x > 0`.

(d) `inc` should increase `n` by exactly one *and* should guarantee that the final value of `n` is strictly positive. If necessary, add preconditions to ensure that it is possible for Alice to achieve this goal.

solution

This changes the postcondition of `inc` to `n > 0 && n == old(n) + 1`. However, for this to be implementable, `inc` should also have a precondition `n >= 0`. (Note that adding this precondition makes the conjunct `n > 0` in the postcondition obsolete).

This restricts Alice again (the implementation from (b) is not acceptable). However, now Bob is also restricted. The following code is not allowed by the new precondition anymore:

```
c.set(-2); c.inc();
```

On the other hand Alice also gains some flexibility! For example, one possible implementation of `inc` which would have not been valid before is:

```
if(n > -10) n = n + 1;
```

Bob also gains some flexibility. Bob's code from case (b) guarantees the postcondition `x == 5`.

Task 3

(from a previous exam)

A `modifies` clause is a part of a method specification that declares which heap locations can be modified by the method. As an example, in the following Java code, method `foo` is not allowed to write to any heap location besides `this.x` and `other.x`.

```
class X {
  public int x;
  public int y;

  /// modifies (this.x, other.x)
  void foo(X other) { ... }
}
```

A `modifies` clause can be interpreted as an implicit postcondition that states that all heap locations that are not listed will have the same value after the method invocation as before.

1. When overriding a method, should it be allowed to *expand* the `modifies` clause, i.e., to include more modifiable heap locations? If yes, just write "yes", otherwise provide a code example that demonstrates the problem when overriding method `foo` with an expanded `modifies` clause.

solution

No.

```
class Y extends X {
  /// modifies (this.x, other.x, this.y)
  void foo(X other) { y = 2; }
}
```

```
public void client() {
  X x = new Y();
  x.y = 12;
  x.foo(x);
  assert x.y == 12;
}
```

2. When overriding a method, should it be allowed to *reduce* the `modifies` clause, i.e., to include less modifiable heap locations? If yes, just write "yes", otherwise provide a code example that demonstrates the problem when overriding method `foo` with a reduced `modifies` clause.

solution

Yes.

Task 4

(from a previous exam)

Assume we add an `otherwise` clause to method contracts in Java, which gives a condition on the state after the method throws an exception. The implementation of the method has to guarantee that the condition in the `otherwise` clause is true whenever the method returns exceptionally (that is, via throwing an exception).

Consider a class with an integer field `f` and the following Java method and its precondition and an `otherwise`-clause (reminder: `final` parameters cannot be assigned to):

```
/// requires  n > 0
/// otherwise f < 0
void foo(final int n) throws IOException
```

Assume method `foo` is overridden in a subclass. Which of the following functions ...

1. ... override `foo` correctly based on the variance rules of Java *and*
2. ... have preconditions and `otherwise`-clauses that would be allowed if the subclass should be a behavioral subtype?

For this, decide what kind of relationship between `otherwise`-clauses of super and subclass should exist, basing your decision on the substitution principle.

For this exercise, assume `FileNotFoundException <: IOException <: Exception` and that there is no integer overflow.

(a)
```
requires  n == 0
otherwise f == -1
void foo(final int n) throws FileNotFoundException
```

(b)
```
requires  n > 0
otherwise f * f > 0
void foo(final int n) throws IOException
```

(c)
```
requires  n >= 0
otherwise f < -n
void foo(final int n) throws Exception
```

(d) **CORRECT:**

```
requires  n != 0
otherwise f == -n
void foo(final int n) throws IOException
```

(e) None of the above would be allowed

Would your answer be the same if `n` were not `final` ?

solution

Exceptions have to be covariant, so (c) cannot be the right answer. `otherwise`-clauses should respect the same rules as postconditions. Namely, overriding methods of subtypes may have stronger `otherwise`-clauses than corresponding supertype methods.

	$\text{Pre}_{super} \Rightarrow \text{Pre}_{sub}$	$old(Pre_{super}) \Rightarrow (Post_{sub} \Rightarrow Post_{super})$	Behavioral subtyping
a	no	yes	no
b	yes	no	no
d	yes	yes	yes

If `n` were not `final` , the right answer would still be (d). Occurrences of a parameter in a postcondition always refer to the value that was originally passed. As such, it does not make any difference if `n` is not `final`.

Task 5

Assume a language with structural subtyping, contravariant arguments, and covariant return types. Is it possible to create the classes `A`, `B`, and `C` that meet all of the following requirements?

1. `B` is a structural subtype of `A`, and `C` is a structural subtype of `B`.
2. `B` is not a behavioral subtype of `A`.
3. `C` is a behavioral subtype of both `A` and `B`.
4. The signatures of any two methods of `A`, `B`, or `C` should be different. For this exercise the signature is the combination of return type, method name, and argument order and types. Note that different signatures do not preclude structural subtyping.
5. The classes `A`, `B`, and `C` do not have any fields.

If it is possible to meet all the above requirements, write the classes `A`, `B`, and `C`.

If it is not possible to meet all the requirements, explain why not. Then pick a requirement and remove it. Write down the classes `A`, `B`, and `C` that meet the remaining four requirements.

In both cases specify the behavior of the classes using contracts. You do not need to provide method bodies. You may use existing Java classes in your solution, if you want to.

solution

All requirements can be met. Here are the corresponding classes:

```
class A {

   /// requires a > 0
   /// ensures result > 0
   Number foo(Integer a)
}

class B {
```

```
    /// requires a > 10
    /// ensures result > 0
    Number foo(Number a)
}

class C {
    /// requires true
    /// ensures result == 10 \/ result == 20
    Integer foo(Object o)
}
```