ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Exercise 3

Subtyping and Behavioral Subtyping

October 11, 2013

In-class Assessment: A subset of the questions from this sheet will be used for the in-class assessment. No notes are allowed during the assessment.

Task 1

Alice and Bob are two software developers. Alice is writing a small class `Cell` that stores one integer. The class supports methods for setting/getting/increasing the integer. Bob is going to write software 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
  public int get() { ... }

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

In the following exercise we will experiment with changing the specifications. In particular, if we change a specification, this might become

- *More restrictive* for a party. For example, a specification that is more restrictive for Alice might 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. If a specification S is more flexible than a specification S' for a party P, then 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 change the postcondition of `get` to:

```
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:

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

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

(c) `inc` should increase `n` by any amount, but it should guarantee that the final value of `n` is positive

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

do the following:

(i) Write formally the new pre/postcondition(s). 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.

Task 2

In this question, we are in a nominal subtyping setting.

Some languages have a special type `MyType` that represents the *dynamic type* of object `this`.

(a) Consider the following code:

```
class Point
{
   int x,y;
   boolean equals(MyType other) { return x==other.x && y==other.y; }
}

class ColorPoint extends Point
{
   int color;
   override boolean equals(MyType other)
   { return super.equals(other) && color==other.color; }
}
```

This definition demands that the dynamic type of the parameter of `equals` is a subtype of the dynamic type of `this`.

Consider the following definitions that give static types to some variables:

```
Point p;
ColorPoint cp1, cp2;
```

and the following calls:

```
p.equals(cp1)    // A
p.equals(cp2)    // B
cp1.equals(p)    // C
cp2.equals(cp1)  // D
cp1.equals(cp2)  // E
```

Assume a sound, statically-checked type system. Which of the calls above must be forbidden and which may be allowed? Why?

(b) Answer the same question, assuming that `ColorPoint` is *final*, i.e., we may not declare new classes as its subtypes.

(c) Assume now that the language includes the feature of *exact types*. An exact type is written `@C` where `C` is a normal type. When we declare that an object `o` is of type `@C`, then `o` is of type `C`, but does not belong to any of the other subtypes of `C`. Change the definitions of our variables as follows

```
@Point p;
@ColorPoint cp1;
ColorPoint cp2;
```

and do not assume that `ColorPoint` is final. Which calls should be forbidden now? Why?

Hint. The classes shown here may be subclassed in code that is not available. The type-checker *cannot* make the assumption that there are no other class definitions elsewhere.

Task 3

Consider the following declarations in Java:

```
interface List
{
   int getSize();
}

interface Iterator
{
   boolean done();
   int getCurrent();
   void next();
   void attach(List l);
}
```

`List` represents sequences of integers and `Iterator` represents a specific traversal of a list. An implementation of an iterator starts iterating over the elements of a list by first calling method `attach`. The following example prints all the elements found during the iteration:

```
void foo(Iterator iter, List list)
{
   iter.attach(list);
   while(!iter.done())
   {
      print(iter.getCurrent());
      iter.next();
   }
}
```

Does `foo` typecheck in Java?

Suppose that we want to have different implementations of lists. For example a linked list and an array are two different ways to implement the `List` interface. What problem would that

cause to the implementers of the iterators? What problem would that cause to the method `foo`?

Task 4

Assume the following class definitions in a nominally typed language:

```
class A {...}
class B extends A {...}
```

Consider now the following two classes:

```
class Super
{
  B foo(B x) { return x; }
}

class Sub extends Super
{
  A foo(A x) { return x; }
}
```

This subtyping is illegal, according to one of the rules presented in (Lecture 2, Slides 22-28). Which one?

However, considering the substitution principle (Lecture 2, Slide 19), this subtyping is *safe*. Why?