ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Exercise 3

Subtyping and Behavioral Subtyping

October 10, 2014

Task 1

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.

Solution

(a) All calls are potentially unsafe and should be forbidden. The reason is that the dynamic type of both the receiver and the parameter are unknown and are not guaranteed to match the restriction that the dynamic type of the parameter should be a subtype of the dynamic type of the receiver.

(b) In this case, we know that the dynamic types of both `cp1` and `cp2` are `ColorPoint`. This guarantees that the calls D and E are ok. However, the first three calls remain unsafe. The first two calls are unsafe because the dynamic type of `p` may be of a subtype of `Point` that has no relation to `ColorPoint`. Call C is not safe, because `p` may be of dynamic type `Point`.

(c) All is known about the dynamic types of `cp1` and `p`. The calls A, B, and E are safe. D is not, because `cp2` may belong to a proper subtype of `ColorPoint`. C is not, because `p` is of dynamic type `Point`.

Task 2

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`?

Solution
The code typechecks.

According to the given specification, an iterator must be able to work with all possible implementations of lists. This is impossible to achieve without downcasting to specific implementations, given that the interface `List` is so small. This solution is also impossible, since not all implementations of `List` are necessarily known at any given point in the program.

A possible problem for `foo` is an implementation of `Iterator` that throws an exception for unknown list implementations.

Task 3

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?

Solution
The interface of `Sub` breaks the overriding rule of Slide 26 (co-variant results).

An object of `Sub` has a wider interface (*it applies to arguments of more types*). When it is applied to an object of type `B`, it happens to return a `B`, which is exactly the behavior expected by an object of `Super`.

Task 4

Let `C` be a class with an integer field `x` and a method `m`. Let `m` have

- Precondition `x>0`
- 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 behavioral subtype of `C`?

a) Pre `x>0` Post `x<-1`

b) Pre `x>0` Post `x<2`

c) Pre `x>-1` Post `x<1`

d) Pre `x>2` Post `x<1`

e) Pre `x>-4` Post `x<-old(x)*old(x)`

f) Pre `true` Post `false`

Solution

	$\text{Pre}_{super} \Rightarrow \text{Pre}_{sub}$	$\text{Post}_{sub} \Rightarrow \text{Post}_{super}$	Behavioral subtyping
a	yes	yes	yes
b	yes	no	no
c	yes	yes	yes
d	no	yes	no
e	yes	yes	yes
f	yes	yes	yes

Task 5

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.

Solution

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

does not guarantee postcondition `x==-1` anymore.

Note: If the implementation of `inc` contains calls to `set`, then there is code on Alice's part that is not valid anymore! For example, the call `set(n+1);` is not an acceptable implementation of `inc` anymore.

(b) This conjoins postcondition `n==old(n)+1` to `inc`. Alice is more restricted: she cannot do this anymore:

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

Bob is more flexible. Now

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

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

(c) This conjoins 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 postcondition `x>0`.

(d) This conjoins postcondition `n>0 && n==old(n)+1` to `inc`. 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 does not guarantee the postcondition `x>-2` anymore:

```
c.set(-2); c.inc(); x=c.get();
```

On the other hand Alice also gains some flexibility! For example, one possible implementation of `inc` which would not be 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`.

Note: Adding invariants is plausible, but one must be careful not to compromise the satisfiability of the specifications. For example, in (d), one could add the invariant `n>=0`, in which case, the precondition `n>=0` to `inc` is not needed. However, there should be an extra precondition to `set`: `p>=0`, to make the specification satisfiable.

Task 6

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 do not have any fields.

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

If it is not possible to meet all requirements, explain why not. Then pick one 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)
}
```