Concepts of Object-Oriented Programming
AS 2013

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Exercise 2

Types and Subtyping

October 4, 2013

Task 1

Suppose that we have a language with structural subtyping, contravariant parameter types and covariant return types. Consider the following types:

```
class A { int m(int x){...}; }
class B { int m(int x){...};  int n(int x) {...}; }
class C { int n(int y){...};  int m(int x) {...}; }
class D { C m(A a) {...}; }
class E { C m(B b) {...}; }
class F { A m(B e) {...}; }
class G { B m(C e) {...}; }
class H { G m(D d, E e) {...}; }
class I { F m(E e, D d) {...}; }
class J { A a; }
class K { B b; }
```

Find all the subtyping relations among them. Assume that `int` has no subtype other than itself.

Solution

`B=C <: A` and `D <: E = G <: F`
No other subtyping relations exist, except the reflexive and transitive closure of the above.

Task 2

As you have seen in the lectures, arrays are covariant in Java and C#. Because of this, each array update requires a run-time type check. Another approach would have been to adopt contravariant arrays. Does this solution require run-time type checks? If this is the case, explain in which cases you need these run-time type checks and provide an example in which a check would fail.

Solution

Yes, contravariant arrays would require run-time type checks when reading values from the array.

By the definition of contravariance, we have that `S<:T` implies `T[]<:S[]`.
Therefore `Object[]<:String[]` since `String<:Object`. So we can pass an array of type `Object[]` to a method that requires a `String[]` argument.

```
class C {
   String foo(String[] a) {
      return a[0];
   }
}
```

```
void client(C c) {
   Object[] arr = new Object[1];
   arr[0] = new Object();
   String s = c.foo(arr);
}
```

Task 3

Consider the following Java program:

```
class B {
   protected int get() {...}
}

class A extends B {
   private int get() {...}
}

class C extends B {
   public int get() {...}
}
```

When we compile it, we obtain the following error:

```
get() in A cannot override get() in B; attempting to
assign weaker access privileges; was protected
        private int get() {...}
                    ^
```

Explain why this is the behavior of the Java compiler.

Solution

Class `A` restricts the accessibility of method `get()`, since it is protected in `B` and private in `A`. This means that class `A` allows fewer behaviors than `B`, so it cannot be a subtype of `B`. On the other hand, class `C` relaxes the accessibility level of method `get()`, so it allows more behaviors than `B`, and this is allowed by the Java compiler.

In general, a class can be a subtype of another class if it assigns "weaker" accessibility permissions that the ones of the superclass.

In Java, there are four different types of access modifiers for fields and methods:

- `public`: every class can access the element
- `protected`: only subclasses and classes in the same package can access the element
- *default*: only classes in the same package can access the element
- `private`: only this class can access the element

We can state that
`public <: protected <: default <: private`

where `a <: b` means that the accessibility level `a` is weaker than `b`, and that a subclass can relax the accessibility level `b` with `a`.

Task 4

Show:

- A program that is rejected by a statically typed language but is executed without typing errors in a dynamically typed language.
- A program that is rejected by a statically typed language and runs into a type error when executed in a dynamically typed language.

Solution

Consider

```
if (x==x) y=1; else y=true;
y = y+1;
```

A usual static type system would reject this program, while the program would not cause typing problems. The static type system would reject the following program which would generate a runtime type error:

```
void f(x) { return x+1 }
print( f(true) )
```

Task 5

Suppose that we have a C#-style programming language supporting the following kinds of parameters:

- "`in`" parameters. The caller provides an expression that is passed by value to the formal parameter.
- "`out`" parameters. The caller provides a variable as the actual parameter. An output from the method is returned and written to the actual parameter when the method terminates. The method does not read the initial value of the actual parameter and the actual parameter has no connection to the formal parameter during the execution of the method.
- "`in out`". The same as "out" but the method may also read the initial value of the actual parameter.
- "`ref`". The parameter is passed by reference. The caller provides a variable that is aliased by the formal parameter during the execution of the method.

What should be the variance rules for these kinds of parameters? Why? Refer to the contravariance rule for method parameters and the covariance rule for return values to explain your answer. Give examples that would not work, if your rules are not followed.

Solution

- "`in`" parameters - contravariant. These parameters are identical to normal parameters in C#. See slide 25.
- "`out`" parameters - covariant. These parameters are essentially named return values, thus the rules for result types apply. See slide 26.
- "`in out`" and "`ref`" parameters - invariant. These can be seen as normal C# parameters coupled with named return values such that their type is always the same. The only way to simultaneously follow the subtyping rules for both is to stay invariant.

Notice that the answer depends on whether a type refers to a value that can be read and/or written by the method. This means that "`in out`" and "`ref`" behave similarly as far as the present question is concerned.

Task 6

Consider the following C# classes:

```
class Point {
   public int x, y;
   virtual public Boolean isEqual(Point p)
   {
      return p.x == x && p.y == y;
   }
};

class ColoredPoint : Point {
   public int color;
   override public Boolean isEqual(ColoredPoint p)
   {
      return p.x == x && p.y == y && p.color == color;
   }
};
```

The compiler refuses to compile this code. Why? Is this reasonable?

What would happen if we wrote the same example in Eiffel? Is there any problem?

What would happen if we removed the `override` keyword? Is there a problem now? (Note that the `override` keyword is mandatory if we want to override a method in C# and that overloading of methods is permitted.)

What would happen if we wrote the same example in Java?

What would happen if we removed the requirement that `ColoredPoint` is a subtype of `Point`?

Solution

The code tries to override a non-existing method. The new method has type `ColoredPoint->bool` and the old method has type `Point->bool`. Since C# classes are invariant in the method parameter types, the new method cannot override the old one. This is reasonable, because the requirement that `ColoredPoint` is a subclass of `Point` entails the following substitution principle: every object of `ColoredPoint` should be useable wherever a point is expected. The substitution principle is not respected whenever a `ColoredPoint c` is compared to a `Point p`, as in `c.isEqual(p)`.

The Eiffel type system tries to accommodate a kind of covariant arguments, but is neither sound nor complete:

```
local
   p: POINT
   c: COLOREDPOINT
   x: POINT
   y: POINT
   r: BOOLEAN
do
   create p
   create c

   x := p
   r := c.isEqual(x) --(1) Fails to compile and would fail at run-time
   x := c
   r := c.isEqual(x) --(2) Fails to compile, but would be ok at run-time
   y := p
   r := x.isEqual(y) --(3) Fails at run-time
end
```

- (1) shows where the type system works: the statement is rejected by the compiler and would fail at runtime (call a non-existing method on an object).

- (2) shows incompleteness: this statement would not compile, but would not fail at runtime. Note that virtually all interesting types systems are incomplete - there are several reasons for this - some type systems are just badly designed, or as in Eiffel try to compromise usefulness, complexity and safety, but in general subtyping in the more expressive type systems is not decidable or is of too high a complexity to be useful so some engineering compromise is usually taken. In most languages we can use an explicit down-cast to bypass the type system, which has the advantage that we know exactly where we have a potential safety issue.
- (3) shows unsoundness: the statement will compile but have a type error at runtime as the actual type of the argument to `x.isequal()` is `POINT` but a `COLOREDPOINT` is required. Eiffel proposes a non-modular (program wide) check that can prevent these cases, but it does not scale well to large programs and is still incomplete - it will reject many valid calls.

If we removed the `override` keyword, the program would compile. Due to overloading, `ColoredPoint` will be a subtype of `Point`, supporting two different methods:

```
boolean isEqual (Point)
boolean isEqual (ColoredPoint)
```

In Java the same thing would happen. However, a Java programmer used to dynamic dispatch will find the following program surprising:

```
void f ()
{
   ColoredPoint p,q;
   p = new ColoredPoint();
   p.x = 1; p.y = 2; p.color = 3;
   q = new ColoredPoint();
   q.x = 1; q.y = 2; q.color = 4;
   boolean b1 = p.isEqual(q); // b1 == false
   boolean b2 = g(p, q); // b2 == true
}

boolean g (ColoredPoint pp, Point qq)
{
   return pp.isEqual(qq); // returns true
}
```

If we don't want `ColoredPoint` to be a subtype of `Point`, we are free to ignore the comparison between the two. However, a language with only subclassing, like Java or C#, will force us to rewrite all the members that could have been reused (in this example, these are only `x`, `y`, but in general, this may be a huge rewriting). Languages that decouple subtyping from inheritance, like C++ and Eiffel, do not have this problem.

Task 7

In C++ object aliasing is achieved using pointers and it is possible to have a pointer to a pointer. Here is an example

```
class X {};

class Initializer {
   public:
      void init(X** x) {
         *x = new X();
      }
};
```

```
class Value {
   private:
      X* x = nullptr;
   public:
      Value(Initializer* i) {
         i->init(&x); // The initializer object will set the value of x
      }
};
```

How does the substitution principle apply to values of type pointer to pointer? Is it safe to call methods that have the signature of `init` with a value of type pointer to pointer to a subtype/supertype of X? Why?

Solution

It is not safe to call methods with the signature of `init` with anything but a pointer to pointer to X. A pointer to a pointer can be thought of as an array with one object. As we know statically safe arrays are invariant. The code below illustrates what might go wrong if the actual argument's type were allowed to vary.

```
class SuperX {};
class X : public SuperX{};
class SubX : public X{};

class Initializer {
   public:
      void init(X** x) {
         *x = new X();
      }
};

class Value {
   private:
      X* x = nullptr;
      SuperX* super_x = nullptr;
      SubX* sub_x = nullptr;
   public:
      Value(Initializer* i) {
         i->init(&x);       // ok
         i->init(&super_x); // wrong, init might want to read super_x
         i->init(&sub_x);   // worng, sub_x might get a value of type X
      }
};
```