Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Exercise 2

Types and Subtyping

October 3, 2014

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

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 3

Show:

- A program that is rejected by a statically typed language but is executed without typing errors 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.

- A program that is rejected by a statically typed language and runs into a type error when executed in a dynamically typed language.

solution

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 4

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 the variance rules for these kinds of parameters be? 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 26.
- "`out`" parameters - covariant. These parameters are essentially named return values, thus the rules for result types apply. See slide 27.
- "`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.

Could "`in out`" and "`ref`" be subtypes of each other? Explain your answer.

solution

"`in out`" and "`ref`" could be made subtypes as far as type errors are concerned - exchanging them could not cause a type error. However, especially in code with concurrency or callbacks, they behave differently and the set of behaviours of neither of them is a subset of the other's behaviours.

Task 5

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{public: int a;};
class SubX : public X{public: int b;};

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

class Initializer2 : public Initializer{
   public:
      virtual void init(X** x) {
         (*x)->a=5; // run time error if called from i->init(&super_x)
            ;
      }
};

class Value {
   private:
      X*      x       = nullptr;
      SuperX* super_x = new SuperX();
      SubX*   sub_x   = nullptr;
   public:
      Value(Initializer* i) {
         i->init(&x);       // ok
         i->init(&super_x); // wrong, if i is of type *Initializer2
         i->init(&sub_x);   // wrong, sub_x might get a value of type
            X
         sub_x->b = 5;      //   and so cause here a run time error
      }
};
```

Task 6 Union Types

Assume a language with nominal subtyping, covariant return types and contravariant arguments that allows types to be defined as a disjunction of other types, as in the following declarations:

```
String || Number get();
void set(String || Number newValue);
```

Such a type is called a *union type* and the different types that form the disjunction are its *components*. Classes can be thought of as union types with just one component.

A type `Sub` is a subtype of another type `Super`, i.e. `Sub <: Super`, if for each component C_{sub} of `Sub` there exists a component C_{sup} of `Super` such that $C_{sub} <: C_{sup}$. The usual nominal subtyping rules apply for classes.

A) Consider the signatures of the four methods below, assuming that `C <: B <: A` (`A`, `B`, and `C` are regular class types)

```
m1: B            foo (B b)
m2: A            foo (A || B ab)
m3: B || C       foo (A a)
m4: A || B || C  foo (C c)
```

Your task is to complete the table below. For each row and column, write 'yes', if the method at the left of the row could override the method at the top of the column. Otherwise write 'no'.

	m1	m2	m3	m4
m1	yes			
m2		yes		
m3			yes	
m4				yes

solution

Note that here `A || B || C`, `A || B`, and `A` can be considered to be identical types, because `C <: B <: A`. Similarly `B || C` can also be considered identical to `B`.

	m1	m2	m3	m4
m1	yes	no	no	yes
m2	no	yes	no	yes
m3	yes	yes	yes	yes
m4	no	no	no	yes

B) Assume that `A`, `B`, and `Q` are classes such that `B <: A` and `Q` is unrelated to `A` and `B`. Consider this code fragment:

```
void foo(A || Q arg) { arg.bar(42); }
```

(i) Assume that the type checker admits method `foo` if all components of `arg`'s static type have a method `bar(int)`. Do we need any run-time checks in order to avoid run-time errors? If so, what are they? Under what conditions could they be omitted?

solution

No run-time checks are necessary.

(ii) Assume that the type checker admits method `foo` if at least one component of `arg`'s static type has a method `bar(int)`. Do we need any run-time checks in order to avoid run-time errors? If so, what are they? Under what conditions could they be omitted?

solution

We could do an additional static check that all components of the type of `arg` have a method `bar(int)` and, if it fails, we need a dynamic check that the run-time type of `arg` has `bar(int)`.

Answer the questions from (i) and (ii) for the code fragment below.

```
void foo(A || B arg) { arg.bar(42); }
```

solution

(i) No run-time checks are necessary.
(ii) We can do a static check that `A` has `bar(int)`. If it does, we do not need a run-time check. Otherwise at run-time we need a check that the type of `arg` is `B`.