Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Exercise 6

Multiple Inheritance, Multiple Dispatch and Linearization

November 4, 2016

Task 1

Consider the following C++ program:

```
class X {
   public:
      X(int p) : fx(p) {}
      int fx;
};
class Y {
   public:
      Y(int p) : fy(p) {}
      int fy;
};
class B : public virtual X,public Y {
   public:
      B(int p) : X(p-1),Y(p-2){}
};
class C : public virtual X,public Y {
   public:
   C(int p) : X(p+1),Y(p+1){}
};
class D : public B, public C {
   public:
      D(int p) : X(p-1), B(p-2), C(p+1){}
};

int main() {
   D* d = new D(5);
   B* b = d;
   C* c = d;
   std::cout << b->fx << b->fy
             << c->fx << c->fy;
   return 0;
}
```

What is the output of running the program?

(a) `5555`

(b) `2177`

(c) **CORRECT:** `4147`

(d) `7177`

(e) `7777`

(f) None of the above

Task 2

Consider the following C++ code:

```
class Person
{
   Person *spouse;
   string name;

public:
   Person (string n) { name = n; spouse = nullptr; }

   bool marry (Person *p)
   {
      if (p == this) return false;
      spouse = p;
      if (p) p->spouse = this;
      return true;
   }

   Person *getSpouse () { return spouse; }
   string getName () { return name; }
};
```

The method `marry` is supposed to ensure that a person cannot marry him-/herself. Without changing the code above, create a new object that belongs to a subclass of `Person` and marry it with itself.

Hint: use multiple inheritance. Explain exactly what happens.

solution

The following C++ code breaks the invariant:

```
class B : public Person
{ public: B (string n) : Person (n) {} };
class C : public Person
{ public: C (string n) : Person (n) {} };
class D : public B, public C
{ public: D (string n) : B(n), C(n) {} };

void marryMyself ()
{
   D me ("Me");
   B *b = &me;
   C *c = &me;
   b->marry (c);
   if (b->getSpouse ()) cout << b->getSpouse ()->getName ();
}
```

The object `me` contains an object of class `B` and an object of class `C`. The addresses of these objects are different and they are obtained using the assignments to `b` and `c` respectively. During the call `b->marry(c)`, the condition `p == this` compares these two addresses and finds them not equal.

Task 3 (from a previous exam)

Consider the following Java classes:

```
class A {
   public void foo (Object o) { System.out.println("A"); }
}
```

```
class B {
   public void foo (String o) { System.out.println("B"); }
}

class C extends A {
   public void foo (String s) { System.out.println("C"); }
}

class D extends B {
   public void foo (Object o) { System.out.println("D"); }
}

class Main {
   public static void main(String[] args) {
      A a = new C(); a.foo("Java");
      C c = new C(); c.foo("Java");
      B b = new D(); b.foo("Java");
      D d = new D(); d.foo("Java");
   }
}
```

What is the output of the execution of the method `main` in class `Main`?

(a) The code will print A C B D

(b) **CORRECT:** The code will print A C B B

(c) The code will print C C B B

(d) The code will print C C B D

(e) None of the above

Task 4

Consider the following C# classes:

```
public class Matrix {
   public virtual Matrix add(Matrix other) {
      Console.WriteLine("Matrix/Matrix");
      return null;
   }
}

public class SparseMatrix : Matrix {
   public virtual SparseMatrix add(SparseMatrix other) {
      Console.WriteLine("SparseMatrix/SparseMatrix");
      return null;
   }
}

public class MainClass {
   public static void Main(string[] args) {
      Matrix m = new Matrix();
      Matrix s = new SparseMatrix();
      add(m,m);
      add(m,s);
      add(s,m);
      add(s,s);
   }
```

```
    public static Matrix add(Matrix m1, Matrix m2) {
        return m1.add(m2);
    }
}
```

A) What is the output of this program? Please explain.

solution

The output is:

```
Matrix/Matrix
Matrix/Matrix
Matrix/Matrix
Matrix/Matrix
```

In each case, C# statically chooses a method variant based on the static type of the receiver. This is only `add(Matrix other)`. Then at run-time, it will call the most-derived override of this method. However, because C# method arguments are invariant, `add(SparseMatrix other)` is not an override and we always end up calling the method from `Matrix`.

B) Without breaking modularity, change only the body of `MainClass.add` to make it possible to always call the most specific `add` method from the matrix hierarchy.

solution

We could change `MainClass` to the following:

```
public static Matrix add(Matrix m1, Matrix m2)
{
    return (m1 as dynamic).add(m2 as dynamic);
}
```

Now, the initial lookup for a method variant is also done at run-time, based not on the static, but on the dynamic type of the receiver. Thus in the third and fourth case there will be a choice between the two different `add` methods in class `SparseMatrix`. To also enable a dynamic look-up of the most-specific method based on the argument types, we additionally cast the argument as `dynamic`.

Task 5

Java 8 allows interface methods to have a default implementation directly in the interface.

A) What are some advantages of this feature?

solution

An advantage is obviously that default implementations can be reused in multiple classes. Another advantage (and the main reason this feature is added to Java) is that default method implementations will allow interface evolution. Without a default implementation, adding new methods to an interface would break all existing classes that implement that interface, since they do not contain an implementation for the new methods. The new features removes this problem.

B) What could be some problems with this feature? How can they be resolved?

solution

A problem could be inheriting two default implementations of the same method from unrelated interfaces. In that case we will have to either choose which implementation we prefer or write a new implementation that overrides both.

Another issue is that interfaces can now suffer from the fragile base class problem. Compared to the usual issue with normal Java classes, this is even more dangerous for interfaces with default methods, since these methods will mostly call other methods of the interface which are overriden in implementing classes. A very restrictive solution here could be to prohibit calls to other methods of the interface, within the implementation of default methods. Alternatively we can "deal" with the problem just like Java deals with the issue in classes - do nothing and rely on the programmer to be careful.

C) What problems of C++ multiple inheritance are avoided by this new design for Java interfaces?

solution

We still avoid problems with correct initialization of fields of super types, since only one super type (the extended class) can have fields, and we can directly call its constructor. Furthermore there are no problems with field duplication as in non-virtual C++ inheritance.

D) Now suppose that, in addition to method implementations, Java also allowed interfaces to define fields. Interfaces would not have constructors and interface fields would always be initialized with a default value.

solution

This makes multiple inheritance in Java very similar to C++.

- What are some advantages of this feature?

solution

An advantage is that we can also reuse fields. This will enable more methods with default implementations in interfaces which could increase code reuse and reduce the effort required to create new classes.

- Given the restrictions above, are there any problems left with such an implementation of multiple inheritance? If so what are they? Propose a solution for each problem you have identified.

solution

These restrictions are somewhat similar to Scala traits, which also do not have specialized constructors (only a default constructor). In this way we manage to avoid problems with initialization order. However a problem that still remains is: how many copies of a field exist? In particular:

  - A class might implement the same interface multiple times (for example by implementing two different interfaces that are a subtype of the same interface). A solution here might be to only have a single copy of the field (as in C++ virtual inheritance).

– A class might implement two different interfaces that both declare the same field. Here we could either restrict interfaces to defining only private fields (which are invisible to the implementor), or we could require some disambiguation syntax when accessing fields, similar to C++ or the proposed syntax for disambiguating conflicting default methods in Java 8.

Task 6

Consider the following declarations in Scala:

```
class C
trait T extends C
trait U extends C
class D extends C
```

Find all the types that can be created with or without traits, as well as the subtype relations between them.

solution

Let X', Y' be the two base classes from which we derive X and Y by mixing in traits. Let A be the set of all traits mixed in to the first class and B the set of all traits mixed in to the second class. The rule is as follows:

$$X <: Y \text{ if and only if } X' <: Y' \text{ and } A \supseteq B.$$

Note: The above rule applies in our example, but it is not a general rule for subtyping in the presence of traits. Notice that `D with T with U` and `D with U with T` are equivalent types (subtypes of each other)! Since, as we saw, they can describe different behavior, this causes a subtle problem for behavioral subtyping!

Task 7

Consider the following Scala code:

```
class Cell
{
   private var x:int = 0
   def get() = { x }
   def set(i:int) = { x=i }
}

trait Doubling extends Cell
{
   override def set(i:int) = { super.set(2*i) }
}

trait Incrementing extends Cell
{
   override def set(i:int) = { super.set(i+1) }
}
```

A) What is the difference between the following objects?

```
val a = new Cell
val b = new Cell with Incrementing
```

```
val c = new Cell with Incrementing with Doubling
val d = new Cell with Doubling with Incrementing
```

solution

Object `a` behaves like a normal cell. Object `b` is also a cell, but it increases the stored value by 1. The interesting difference is between `c` and `d`. They are both cells. They have mixed in exactly the same traits. However, calling `set(i)` has a different effect on them: it stores `2i+1` to the first one and `2(i+1)` to the second one.

B) We use the following code to implement a cell that stores the argument of the set method multiplied by four:

```
val e = new Cell with Doubling with Doubling
```

Why doesn't it work? What does it do? How can we make it work?

solution

Trait `Doubling` will not get mixed in twice, as perhaps the programmer would expect. Scala rejects this statically.

The problem can be bypassed in an ugly way, by creating a new trait `Doubling2` that behaves exactly like `Doubling` and then introducing `e = new Cell with Doubling with Doubling2`. Here is our first try:

```
trait Doubling2 extends Doubling
val e = new Cell with Doubling with Doubling2
```

The code passes through, but dynamically `e` behaves as if it were a `Cell with Doubling`. Scala lets the code go through, because `Doubling2` may introduce new functionalities, but refuses to include `Doubling` twice in the linearization.

Our last try, the ugliest of all, but the one which will finally work, is to create a whole new trait from scratch, reusing nothing:

```
trait Doubling3 extends Cell
{
   override def set(i:int) = { super.set(2*i) }
}
val e = new Cell with Doubling with Doubling3
```

And now `e.set` quadruples its argument as expected.

C) Find a modularity problem in the above, or a similar, situation. Hint: a client that gets given a class `C` does not necessarily know if a trait `T` has been mixed in that class.

solution

A problem is that a method that accepts `Cell with Doubling with Incrementing` as an argument could also be passed a class of the type `Cell with Incrementing with Doubling` - so what it can actually assume about its inputs is less than would be expected.

D) We propose the following solution to support traits together with behavioral subtyping: Assume `C` is a class with specification `S`. Each time we create a new trait `T` that extends `C`, we must ensure that `C with T` also satisfies `S`.
Show a counterexample that demonstrates that this approach does not work.

solution

Consider the following example:

```
class C
{
   var x:int;
   def foo() = {} //ensures true
}
trait T1 extends C
{
   override foo() = { x=x+1 } //ensures x>old(x)
}
trait T2 extends C
{
   override foo() = { x=x-1 } //ensures x<old(x)
}
```

Both `C with T1` and `C with T2` are behavioral subtypes of `C`. But `C with T1 with T2` is not a behavioral subtype of `C with T1`.

Task 8

Consider the following Scala code:

```
class A           {          def bar() =              ""   }
trait B extends A { override def bar() = super.bar()+"B" }
trait C extends B { override def bar() = super.bar()+"C" }
trait D extends B { override def bar() = super.bar()+"D" }

object Main {
  def main()
  {
    foo(new A with D with C with B())
  }
  def foo(x:A with D)
  {
    println(x.bar())
  }
}
```

What would be the output of the call `Main.main()`?

(a) BDB

(b) BBDBC

(c) BBCBD

(d) DB

(e) **CORRECT:** BDC

(f) BCD

(g) None of the above

Task 9 (from a previous exam)

Consider a language with Java-like syntax and with multiple inheritance. Overriding follows a linearization order which, in this task, proceeds *left to right* within the subclass declaration,

expanding superclasses *depth first* and skipping repeated classes. For example, consider the following classes:

```
class A {
   public void foo() { System.out.println("A"); }
}

class B extends A { }

class C extends A {
   public void foo() { System.out.println("C"); }
}

class X extends B, C { }

class Y extends C, B { }
```

The linearization order for `X` is `X`, `B`, `A`, `C`. Thus calling `foo()` on an instance of `X` prints "`A`". The linearization order for `Y` is `Y`, `C`, `A`, `B`. Thus calling `foo()` on an instance of `Y` prints "`C`".

A) Consider the following class `Z`:

```
class Z extends Y, X { }
```

What is the linearization order for `Z`?

solution

The order is `Z`, `Y`, `C`, `A`, `B`, `X`.

B) We now add a method `qux` to `X`:

```
class X extends B, C {
   public void qux() { bar(); }
}
```

- Add a method `bar` to either `A`, `B`, or `C` such that calling `qux()` on an instance of `X` prints "`A`" and calling `bar()` on an instance of `C` prints "`C`". Do not use casts or type checks.

  Hint. Any dynamically bound method call is resolved searching from the dynamic type of the receiver in the linearization order.

solution

One possible solution is modifying `C` as follows:

```
class C extends A {
   public void foo() { System.out.println("C"); }
   public void bar() { foo(); }
}
```

- What happens in your solution when calling `bar()` on an instance of `B`?

solution

In the case of the previous solution, calling `bar()` on an instance of `B` would raise a compile-time error since instances of `B` do not have a method `bar`.

solution

Another possible solution is modifying `A` as follows:

```
class A {
   public void foo() { System.out.println("A"); }
   public void bar() { foo(); }
}
```

In this case, calling `bar()` on an instance of `B` would print "`A`".

C) With respect to your solution from part **B**, we now modify the method `foo` in `C` as follows:

```
public void foo() { bar(); }
```

- What happens when calling `qux()` on an instance of `X`?

solution

Calling `qux()` on an instance of `X` prints "`A`".

- What happens when calling `bar()` on an instance of `C`?

solution

Calling `bar()` on an instance of `C` does not terminate or raises a runtime error due to infinite mutual recursion.