12 Polymorphism (Continued)

Last time, we saw that we can parameterize both interfaces and methods — today we complete the picture by showing that we can also parameterize classes, naturally enough.

Before doing that, though, let me revisit the issue of information loss we saw back in lecture 8 when we saw the difference between compile-time types and run-time types, and see how polymorphic methods can help us reduce that information loss.

12.1 Information Loss and Polymorphic Methods

Recall the example from lecture 8. We had Point and CPoint, where the latter is a subclass of the former, and we were considering the following pair of functions:

```java
public static Point identity (Point p) {
    return p;
}

public static Point clone (Point p) {
    return Point.create(p.xPos(),p.yPos());
}
```

We saw that these two functions have the same signature — they both take a Point as an argument, and they return a Point. At run time, though, they behave quite differently: while identity returns its argument untouched, clone returns a new Point constructed from its argument.

It is perfectly safe to pass a CPoint to both of those functions, but how does the type system treat the result? We saw that the type system, because it only uses the signature of a function when type checking, cannot distinguish between the above two functions, and therefore to be conservative must assume that the result is always a Point.

We illustrated this using the following function:

```java
public static Color extractColor1 (CPoint cp) {
    ??? newcp = identity(cp);
    return newcp.color();
}
```
We saw that the only thing we can write in place of ?? is Point, which leads to the type system rejecting the call to newcp.color() because newcp, being considered a Point, is not known to have a color() method at compile time. We’ve lost compile-time type information about the result of identity. Even though we know full well that it should return a CPoint. We know that. The type system doesn’t. To solve the problem, solution 1 was to use a cast:

```java
public static Color extractColor (CPoint cp) {
    Point newcp_temp = identity(cp);
    CPoint newcp = (CPoint) newcp_temp;
    return newcp.color();
}
```

But that’s annoying. The cast is a hack to compensate for the loss of information. Now that we have polymorphic functions, though, we can do much better. We can give a more precise type to identity(). The problem with the current type for identity() is that it is not precise enough: it says that it accepts a Point (or one of its subclasses) and returns a Point. That’s not precise enough because we know that at run time, the type of the result is always the type of its argument. Polymorphic function let us write exactly that:

```java
public static <T extends Point> T identityBetter (T p) {
    return p;
}
```

Read this out: for any type T that happens to be a subclass of Point, identity can take a value of type T and return a value of that exact type T. That’s much better. We can now write:

```java
public static Color extractColor (CPoint cp) {
    CPoint newcp = identityBetter(cp);
    return newcp.color();
}
```

Now that type system looks at identityBetter, sees that it is polymorphic, determines what type you want identityBetter at, in this case from the argument of the call, which is CPoint, so it sees if it can type checks the call taking T=CPoint in the signature of identityBetter, which then says that the result has type CPoint, which agrees with the declared type CPoint of newcp, and everything works great. No need for a cast, because we’ve managed to give a better, more precise type to the function.

## 12.2 Polymorphic Classes

We now turn to the final bit of parameterization that I have not yet mentioned, polymorphic classes. This would be useful certainly for lists, which should be defined for some type T of
underlying values, instead of fixing a type such as Integer or AInteger in the definition. A class can be parameterized just like an interface, using a similar declaration.

### 12.2.1 Polymorphic Lists

Here is the definition of List<A>, a parameterized version of List, with the following signature, and the same specification as earlier — we may call this a parameterized or a polymorphic ADT:

**CREATORS** List<A> empty ()  
List<A> cons (A, List<A>)

**ACCESSORS** boolean isEmpty ()  
A first ()  
List<A> rest ()  
String toString ()

with the expected specification. I’m dropping functional iterators for now.

Following the design pattern we have for deriving an implementation from an ADT, we get the following code:

```java
/* ABSTRACT CLASS FOR LISTS */
public abstract class List<A> {

    public static <B> List<B> empty () {
        return new EmptyList<B>();
    }

    public static <B> List<B> cons (B i, List<B> l) {
        return new ConsList<B>(i,l);
    }

    public abstract boolean isEmpty ();

    public abstract A first ();

    public abstract List<A> rest ();

    public abstract String toString ();
}
```
/ * CONCRETE CLASS FOR EMPTY CREATOR */

class EmptyList<A> extends List<A> {

    public EmptyList () {}

    public boolean isEmpty () {
        return true;
    }

    public A first () {
        throw new Error("EmptyList.first()");
    }

    public List<A> rest () {
        throw new Error("EmptyList.rest()");
    }

    public String toString () {
        return ";";
    }
}

/* CONCRETE CLASS FOR CONS CREATOR */
class ConsList<A> extends List<A> {

    private A firstElement;
    private List<A> restElements;

    public ConsList (A f, List<A> r) {
        firstElement = f;
        restElements = r;
    }

    public boolean isEmpty () {
        return false;
    }

    public A first () {
        return firstElement;
    }
}
public List<A> rest () {
    return restElements;
}

public String toString () {
    return firstElement.toString() + " " + restElements.toString();
}

A few things to note. First, the definition/use distinction: the declaration `class EmptyList<A>...`
defines the type variable `A`, while everything else is a use of `A`. Once the type checker figures out the type for `A` that you want, every use of `A` is taken to be that type. Thus, for instance, in `class EmptyList<A> extends List<A>`, the second `<A>` is a use, it says that `EmptyList<A>` that you create extends `List<A>` for that exact same `A` that you need. Thus, if `A= Integer` because you’re creating an instance of `EmptyList` at type `Integer` — by invoking the appropriate constructor, see below — the resulting `EmptyList<Integer>` will be a subclass of `List<Integer>`.

Second, to invoke a constructors for a polymorphic class, you need to supply the type at which you want to instantiate the class. For instance, `new EmptyList<Integer>().` If you forget that, the system will use `Object` as a type, which generally will not do what you want.

The other thing to notice is that static methods are polymorphic. That’s needed because of the way Java deals with polymorphic classes. More specifically, the way Java deals with type parameters — they are technically associated with an instance of a class, and because what is associated with an instance of a class is not accessible from static elements in the class, the static methods in a class cannot refer to the parameter. Meaning that in order to write a creator `cons()` that takes an element of type `B` and a list of `B`s and returns a list of `B`s, we need to say that the type of that creator is: for all types `B`, the `cons` creators takes an `B` and a `List<B>` and produces a `List<B>`, which gives us the above method definitions.\(^1\)

This means, in particular, that the type variable used in a static method has nothing to do with the type parameter in the class implementing that static method. To emphasize this, I usually use a different type variable name in static methods in parameterized classes.

**Easy Exercise:** Add a method `funcIterator()` to the above signature, with type `FuncIterator<A>`

\(\text{funcIterator()}\), where `A` is the type variable for class `List<A>`. Implement those functional iterators — which should be implemented similarly as for class `List` earlier in the

---

\(^1\)Here is an explanation, if you’re curious. In Java, the code for a polymorphic class is not actually duplicated, there is really only one definition of `List` around, and so the type parameter of a polymorphic class is thought of as kind-of-special field, And fields in an object are not visible from static methods in the class. And indeed, you should be able to invoke `empty` even if you have no lists around. Another consequence of the Java handling of polymorphism is that type parameters, as soon as type checking is done, do not actually exist at runtime. This means, in particular, that we cannot use a type argument in places where the type would have a runtime existence, such as in a cast; uses such as `T x = (T) foo` are disallowed, as well as `instanceof` checks.)
Once you’ve defined iterators for polymorphic lists, then you can still reuse the iterator functions we defined last time, like:

```java
List<Integer> nothing = List.empty();
List<Integer> onetwothree = List.cons(1, List.cons(2, List.cons(3, nothing)));

FuncIterator<Integer> iterator = onetwothree.funcIterator();
printElements(iterator);
```

which yields

```
Element = 1
Element = 2
Element = 3
```

as expected.

### 12.3 Polymorphic Pairs

At the end of last lecture, we defined list of pairs of integers, as its own special implementation of lists. We can now define list of pairs by instantiating the polymorphic list implementation above using a type for pairs of integers. In order to do that, we need an ADT for pairs. For generality, let’s define a polymorphic ADT for pairs, the `Pair<T,U>` ADT of pairs of type `T` and `U`, with signature:

CREATORS `Pair<T,U> create (T, U)`

ACCESSORS `T first ()`
`U second ()`
`String toString ()`

with specification:

```
create(f,s).first() = f
create(f,s).second() = s
create(f,s).toString() = "(" + f + "," + s + ")"
```

I’m keeping this ADT very simple. The implementation of the ADT is completely straightforward:
public class Pair<T,U> {

    private T first;
    private U second;

    protected Pair () {}

    private Pair (T f, U s) {
        first = f;
        second = s;
    }

    public static <V,W> Pair<V,W> create (V f, W s) {
        return new Pair<V,W>(f,s);
    }

    public T first () {
        return first;
    }

    public U second () {
        return second;
    }

    public String toString () {
        return "(" + first().toString() + "," + second().toString() + ")";
    }
}

Once you have this, is is a simple matter to define lists of pairs of integers, and if you have correctly defined iterators for lists, you can print all the elements of a list of pairs of integers using the polymorphic printElements function:

    List<Pair<Integer,Integer>> nothing2 = List.empty();
    List<Pair<Integer,Integer>> l2 =
        List.cons(pi(1,2),
                 List.cons(pi(3,4),
                          List.cons(pi(5,6), nothing2)));

    printElements(l2.funcIterator());

where pi() is defined to be:
public static Pair<Integer, Integer> pi(int i, int j) {
    return Pair.create(new Integer(i), new Integer(j));
}

and which outputs:

Element = (1,2)
Element = (3,4)
Element = (5,6)

A nice example of reuse: we've reused the polymorphic class List (by having it implement lists of pairs of integers instead of just lists of pairs), and we've reused the polymorphic function printElements.

### 12.4 Addable Pairs

What about the polymorphic function sumElements? Remember that we wanted to use is to sum the elements provided by an iterator that yields elements of a type that has an add() method defined. We defined a form of integers, called AInteger that implements the Addable interface, and that could be used for this purpose. For instance, we can construct an instance of a list of AInteger (using the polymorphic list implementation), and invoke sumElements on it:

```java
List<AInteger> nothing3 = List.empty();
List<AInteger> l3 = List.cons(AInteger.create(1),
                                List.cons(AInteger.create(2),
                                         List.cons(AInteger.create(3),
                                                   nothing3)));

IteratorLib.printElements(l3.funcIterator());
System.out.println("Sum = " + IteratorLib.sumElements(AInteger.create(0),
                                                       l3.funcIterator()));
```

which outputs:

Element = 1
Element = 2
Element = 3
Sum = 6

So defining lists of AIntegers and calling sumElements works, because AInteger is a subclass of Addable. But what about summing elements of an iterator that returns pairs of integers.
Last time we defined PairAI which implemented the Addable interface by having an add() method. Our pairs, defined above, do not have an add() method — nor should it have one, because an add() method should only make sense if the underlying types of the pair (that is, the type of the first component of the pair and the type of the second component of the pair) themselves have an add() method.

The solution is to define an AddablePair, that is a form of Pair (i.e., a subclass), which defines an add() method, as long as its component types have an add() method as well — that is, as long as its component types are subclasses of Addable. The way to do that is to define the AddablePair class to be parameterized, and to put constraints on those bounds, a bit like when we put constraints on the type of a polymorphic function, in sumElements for example. Here is the implementation of AddablePair:

```java
public class AddablePair<T extends Addable<T>, U extends Addable<U>> extends Pair<T,U> implements Addable<AddablePair<T,U>> {
    private T first;
    private U second;

    private AddablePair (T f, U s) {
        first = f;
        second = s;
    }

    public static <V extends Addable<V>, W extends Addable<W>>
    AddablePair<V,W> create (V f, W s) {
        return new AddablePair<V,W>(f,s);
    }

    public T first () {
        return first;
    }

    public U second () {
        return second;
    }

    public AddablePair<T,U> add (AddablePair<T,U> p) {
        return create(first().add(p.first()),second().add(p.second()));
    }

    public String toString () {
```
Look at the declaration of the class, and make sure you understand what it is saying. It is saying a lot, and everything it is saying is important: it says that `AddablePair` is a polymorphic class that is a subclass of `Pair`, it is parameterized by two types, those two types better be subclasses of `Addable`, and the resulting class `AddablePair` itself is also a subclass of `Addable`.

This means that if we define a list that stores elements of type `AddablePair`, then an iterator for that list can be given to `sumElements()` so that it can sum its elements. Let’s define a sample list of pairs of `AInteger`s, which are `Addable` as we already saw:

```java
List<AddablePair<AInteger, AInteger>> nothing4 = List.empty();
List<AddablePair<AInteger, AInteger>> l4 =
    List.cons(pai(1, 2),
              List.cons(pai(3, 4),
                        List.cons(pai(5, 6), nothing4)));
```

```java
IteratorLib.printElements(l4.funcIterator());
System.out.println("Sum = " + IteratorLib.sumElements(pai(0, 0),
                                                      l4.funcIterator()));
```

where `pai()` is a helper function to create addable pairs of integers:

```java
public static AddablePair<AInteger, AInteger> pai (int i, int j) {
    return AddablePair.create(AInteger.create(i), AInteger.create(j));
}
```

and the result is the output:

```
Element = (1,2)
Element = (3,4)
Element = (5,6)
Sum = (9,12)
```

Again, we’ve managed to reuse a lot of code. And now that we have the notion of an addable pair, we can construct more complex addable pairs by pairing addable stuff, for instance, we can construct an instance of `AddablePair<AInteger, AddablePair<AInteger, AInteger>>`, which are addable pairs of an addable integer and an addable pair of addable integers — in other words, a triple of addable integers. Make sure you understand how addition for such triples would work.