13 Examples: Iterator Gadgets

Today, I want to look at a few more examples of the kind of things you can express with functional iterators and polymorphism.

The idea that I want to propose is to consider that a functional iterator is a way to produce elements (the exact kind of elements produced depending of course on the type of the iterator). Putting it another way, a functional iterator is a standardized interface to something that can produce elements. Having a standardized interface means that as soon as you have a gadget that can connect to that standardized interface, you can connect it to anything that implement that standardized interface. If that gadget itself produces elements according to that standardized interface, you have the beginning of a system that lets you connect gadgets together in a standardized way. In the case of functional iterator, they produce elements. You can imagine connecting gadgets to those iterators that modify the elements produced before passing them on, or combining the elements from two iterators into a single element. Those gadgets for all intents and purpose can be made to look like functional iterators to the outside world.

Let me illustrate this with a simple example. Suppose we want to write a little gadget that you can hook to a functional iterator that produces integers, and that simply doubles the elements produced by the iterator. That gadget, once you connect it to the functional iterator, itself looks like a functional iterator: if you ask it for an element, it will ask the iterator it is connected to for an element, double it, and pass it on. If you ask it to move to the next element, it will update the iterator it is connected to make it point to the next element, and so on.

Here is a simple `DoubleGadget` that does the work.

```
public class DoubleGadget implements FuncIterator<Integer> {

  private FuncIterator<Integer> connectedTo;

  private DoubleGadget (FuncIterator<Integer> it) {
    connectedTo = it;
  }

  public static DoubleGadget create (FuncIterator<Integer> it) {
    return new DoubleGadget(it);
```

```
    }

    public boolean hasElement () {
        return connectedTo.hasElement();
    }

    public Integer element () {
        return 2*connectedTo.element();
    }

    public FuncIterator<Integer> moveToNext () {
        return DoubleGadget.create(connectedTo.moveToNext());
    }

}
```

You create a `DoubleGadget` by calling the `create()` creator, passing it a functional iterator to connect to. Note the `element()` method, that simply doubles the value obtained from the iterator the gadget is connected to, and the `moveToNext()` method, that moves the iterator the gadget is connected to to its next element, and creates a new `DoubleGadget` connected to that new iterator.

Before we look at a simple example of how we can use `DoubleGadget`, let's define a couple of functional iterators to serve as the basis for all our examples.

```
List<Integer> nothing = List.empty();
List<Integer> a = List.cons(1, List.cons(2, List.cons(3, nothing)));
List<Integer> b = List.cons(66, List.cons(99, List.cons(89,
                        List.cons(11, nothing))));
```

I'm assuming we have a function `show()` defined as:

```
public static <B> void show (String s, FuncIterator<B> it) {
  FuncIterator<B> temp = it;

  System.out.print(s + " =");
  while (temp.hasElement()) {
    System.out.print ("  " + temp.element());
    temp = temp.moveToNext();
  }
  System.out.println("");
}
```

so that:

```
show("a", a.funcIterator());

show("b", b.funcIterator());
```

outputs

```
a =  1  2  3
b =  66  99  89  11
```

All nice and simple. Let's try out the `DoubleGadget`:

```
show("Double(a)", DoubleGadget.create(a.funcIterator()));

show("Double(b)", DoubleGadget.create(b.funcIterator()));
```

which creates `DoubleGadget`s connected to the iterator for `a` and `b`, respectively, and show the result of printing all the elements of those iterators:

```
Double(a) =  2  4  6
Double(b) =  132  198  178  22
```

A `DoubleGadget` works only with integer-based functional iterators. Let's write a polymorphic gadget, a `SkipGadget`, that connects to a functional iterator and delivers only one out of every two element produced by that iterator.

```
public class SkipGadget<A> implements FuncIterator<A>  {

   private FuncIterator<A> connectedTo;

   private SkipGadget (FuncIterator<A> it) {
      connectedTo = it;
   }

   public static <B> SkipGadget<B> create (FuncIterator<B> it) {
      return new SkipGadget<B>(it);
   }

   public boolean hasElement () {
      return connectedTo.hasElement();
   }
```

```
    public A element () {
        return connectedTo.element();
    }

    public FuncIterator<A> moveToNext () {
        FuncIterator<A> next = connectedTo.moveToNext();
        if (next.hasElement()) {
            FuncIterator<A> nnext = next.moveToNext();
            return SkipGadget.create(nnext);
        }
        return SkipGadget.create(next);
    }
}
```

As before, we create such a gadget by calling the `create()` creator, giving it a functional iterator to connect to. The invariant that the `SkipGadget` maintains is that it is always pointing to the next element it will produce from the underlying iterator. Thus, `element()` simply returns that element. When `moveToNext()` is invoked, though, we skip the actual next element, and point to the following one. (As before, we actually create a new `SkipGadget` to wrap around the iterator we obtain by moving the iterator we are connected to.) Note that when `moveToNext()` moves past the last element of the iterator (either because we are at the last element or because the element we're skipping is the last element), we return an iterator that has no elements left (and we have to be careful not to throw an exception in that case).

Here is an example using lists `a` and `b`, and also showing that we can connect the `SkipGadget` to a `DoubleGadget`:

```
show("Skip(a)", SkipGadget.create(a.funcIterator()));

show("Skip(b)", SkipGadget.create(b.funcIterator()));

show("Skip(Double(a))",
        SkipGadget.create(DoubleGadget.create(a.funcIterator())));
```

with the expected output:

```
Skip(a) =  1  3
Skip(b) =  66  89
Skip(Double(a)) =  2  6
```

Let's look at a slightly more complex gadget. Suppose we have two functional iterators. We can imagine a gadget that connected those two iterators in sequence: it first delivers all the

elements from the first iterator, and when the elements of that iterator are exhausted, it delivers the elements of the second iterator. Call that a `SequenceGadget`.

```
public class SequenceGadget<A> implements FuncIterator<A> {

   private FuncIterator<A> connectedTo1;
   private FuncIterator<A> connectedTo2;

   private SequenceGadget (FuncIterator<A> c1, FuncIterator<A> c2) {
      connectedTo1 = c1;
      connectedTo2 = c2;
   }

   public static <B> SequenceGadget<B> create (FuncIterator<B> c1,
         FuncIterator<B> c2) {
      return new SequenceGadget<B>(c1,c2);
   }

   public boolean hasElement () {
      return (connectedTo1.hasElement() || connectedTo2.hasElement());
   }

   public A element () {
      if (connectedTo1.hasElement())
         return connectedTo1.element();
      return connectedTo2.element();
   }

   public FuncIterator<A> moveToNext () {
      if (connectedTo1.hasElement())
         return SequenceGadget.create(connectedTo1.moveToNext(),
                                      connectedTo2);
      return SequenceGadget.create(connectedTo1,connectedTo2.moveToNext());
      // alternatively: return connectedTo2.moveToNext();
   }
}
```

When we create a `SequenceGadget`, we give it two functional iterators (that produce the same type of elements). The `SequenceGadget` records where in the first iterator it is pointed, and where in the second iterator it is pointing. When we ask for an element, it returns an element from the first iterator if available, otherwise, from the second. When moving the iterator, it tries to move the first iterator only. If the first iterator is exhausted, then it

moves the second iterator. Think about it some, read the code, meditate on it. Try drawing pictures to understand what is happening. This is a common pattern when working with iterators.

Let's try it out on a variety of examples, including combinations with other gadgets:

```
show("a & b", SequenceGadget.create(a.funcIterator(),
                                    b.funcIterator()));

show("Skip(a) & Double(b)",
        SequenceGadget.create(SkipGadget.create(a.funcIterator()),
                              DoubleGadget.create(b.funcIterator())));

show("a & a", SequenceGadget.create(a.funcIterator(),a.funcIterator()));
```

with outputs:

```
a & b =  1  2  3  66  99  89  11
Skip(a) & Double(b) =  1  3  132  198  178  22
a & a =  1  2  3  1  2  3
```

Let's complicate things even further. Right now, all our gadgets have returned elements of the same type as the elements of the iterator they are connected to. Let's write a gadget that returns something different. In particular, let's write a gadget that connects to two iterators and that returns *pairs* of elements, one taken from each iterator, call it a `ZipGadget`. We need to figure out what happens when one of the iterators the gadget is connected to exhausts before the other does. For simplicity, we make the `ZipGadget` have no more elements when one of the iterators it is connected to has no more elements.

```
public class ZipGadget<A,B> implements FuncIterator<Pair<A,B>> {

   private FuncIterator<A> connectedTo1;
   private FuncIterator<B> connectedTo2;

   private ZipGadget (FuncIterator<A> c1, FuncIterator<B> c2) {
      connectedTo1 = c1;
      connectedTo2 = c2;
   }

   public static <C,D> ZipGadget<C,D> create (FuncIterator<C> c1,
            FuncIterator<D> c2) {
      return new ZipGadget<C,D>(c1,c2);
   }
```

```
    public boolean hasElement () {
        return (connectedTo1.hasElement() && connectedTo2.hasElement());
    }

    public Pair<A,B> element () {
        return Pair.create(connectedTo1.element(), connectedTo2.element());
    }

    public FuncIterator<Pair<A,B>> moveToNext () {
        return ZipGadget.create(connectedTo1.moveToNext(),
                                connectedTo2.moveToNext());
    }
}
```

This code uses the polymorphic class `Pair` I gave you in the last lecture. Again, we create a `ZipGadget` by passing in two functional iterators to connect to. When calling `element()`, we read the element currently pointed to by booth iterators, and create a pair that we return. When calling `moveToNext()`, we simply move each of the underlying iterators, creating a new `ZipGadget` connected to the newly created iterators. There's actually less bookkeeping in `ZipGadget` than in `SequenceGadget`.

Again, some examples:

```
show("Zip(a,b)", ZipGadget.create(a.funcIterator(),b.funcIterator()));

show("Zip(a,Skip(b))",
        ZipGadget.create(a.funcIterator(),
                         SkipGadget.create(b.funcIterator())));
```

with the expected output:

```
Zip(a,b) =  (1,66)  (2,99)  (3,89)
Zip(a,Skip(b)) =  (1,66)  (2,89)
```

`Exercise:` *implement a* ***`SumGadget`****, that connects to two functional iterators that return elements of a type that can be added (i.e., subclasses of* ***`Addable`****, as we saw last lecture), and that produces the sum of the elements produced by the two iterators. Here is a signature and specification for* ***`SumGadget`****, if it helps. The challenge really is in declaring the class correctly to make sure that* ***`add`*** *is available.*

```
public static SumGadget<A> create (FuncIterator<A>, FuncIterator<>A)
public boolean hasElement ()
```

```
public A element ()
public FuncIterator<A> moveToNext ()

create(f,g).hasElement()
  = false   if f.hasElement() = false
  = false   if g.hasElement() = false
  = true    otherwise
create(f,g).element() = f.element().add(g.element())
create(f,g).moveToNext() = create(f.moveToNext(),g.moveToNext())
```

The above gadgets are all fairly straightforward, in the sense that they do not require tricks of any kinds, or any fields beyond the obvious ones, or auxiliary functions or auxiliary classes. The following few examples require a bit more cleverness.

First, let's write a variant of the `ZipGadget` that connects to two functional iterators and returns pairs of elements from those two iterators. Instead of pairing the first elements together, then the second elements together, then the third elements together, etc, we want to produce pairs of all the possible combinations of an element from the first iterator and an element of the second iterator. Call it a `CartesianGadget` (inspired by the Cartesian product of two sets in set theory).

```
public class CartesianGadget<A,B> implements FuncIterator<Pair<A,B>> {

  private FuncIterator<A> connectedTo1;
  private FuncIterator<B> connectedTo2;
  private FuncIterator<B> initConnectedTo2;

  private CartesianGadget (FuncIterator<A> c1, FuncIterator<B> c2,
                           FuncIterator<B> initC2) {
    connectedTo1 = c1;
    connectedTo2 = c2;
    initConnectedTo2 = initC2;
  }

  public static <C,D> CartesianGadget<C,D> create (FuncIterator<C> c1,
                                                   FuncIterator<D> c2) {
    return new CartesianGadget<C,D>(c1,c2,c2);
  }

  public boolean hasElement () {
    return connectedTo1.hasElement();
  }
```

```
    public Pair<A,B> element () {
        return Pair.create(connectedTo1.element(), connectedTo2.element());
    }

    public FuncIterator<Pair<A,B>> moveToNext () {
        FuncIterator<B> nextC2 = connectedTo2.moveToNext();
        if (nextC2.hasElement())
            return new CartesianGadget<A,B>(connectedTo1,nextC2,
                                            initConnectedTo2);
        return new CartesianGadget<A,B>(connectedTo1.moveToNext(),
                                        initConnectedTo2,
                                        initConnectedTo2);
    }
}
```

A `CartesianGadget` is used pretty much like a `ZipGadget`, except that we need to loop through the second iterator for every element produced by the first iterator. In order to do this, we need to remember the initial second iterator that we used when we constructed the `CartesianGadget` in the first place. Note that the constructor for the class takes an extra argument compared to the creator `create()`.

Here's an example:

```
show("Cartesian(a,b)", CartesianGadget.create(a.funcIterator(),
                                              b.funcIterator()));

show("Cartesian(a,a)",
        CartesianGadget.create(a.funcIterator(),a.funcIterator()));

show("Cartesian(a,Zip(b,Double(b)))",
        CartesianGadget.create(a.funcIterator(),
                               ZipGadget.create(b.funcIterator(),
                                                DoubleGadget.create(b.funcIterator()))));
```

with the expected output:

```
Cartesian(a,b) =  (1,66)  (1,99)  (1,89)  (1,11)  (2,66)  (2,99)
  (2,89)  (2,11)  (3,66)  (3,99)  (3,89)  (3,11)
Cartesian(a,a) =  (1,1)  (1,2)  (1,3)  (2,1)  (2,2)  (2,3)  (3,1)
  (3,2)  (3,3)
Cartesian(a,Zip(b,Double(b))) =  (1,(66,132))  (1,(99,198))
  (1,(89,178))  (1,(11,22))  (2,(66,132))  (2,(99,198))  (2,(89,178))
  (2,(11,22))  (3,(66,132))  (3,(99,198))  (3,(89,178))  (3,(11,22))
```

Finally, here is a gadget that connected to a functional iterator, and produces the elements in the reverse order as the underlying iterator: first the last, then the second-to-last, then the third-to-last, etc. There are a few ways of writing such a `ReverseGadget`, but they're all somewhat tricky. Here's an approach that uses two auxiliary classes – an approach very much reminiscent of the implementation of lists we obtained form the Specification design pattern.

```
public abstract class ReverseGadget<A> implements FuncIterator<A>  {

   private static <B> ReverseGadget<B> helper (ReverseGadget<B> r,
                                               FuncIterator<B> it) {
      if (it.hasElement()) {
         ReverseGadget<B> newRev =
            new ReverseGadget.NonEmpty<B>(it.element(),r);
         return helper(newRev,it.moveToNext());
      } else {
         return r;
      }
   }

   public static <B> ReverseGadget<B> create (FuncIterator<B> it) {
      return helper(new ReverseGadget.Empty<B>(), it);
   }

   public abstract boolean hasElement ();
   public abstract A element ();
   public abstract FuncIterator<A> moveToNext ();


   private static class Empty<B> extends ReverseGadget<B> {
      public Empty () {}

      public boolean hasElement () {
         return false;
      }

      public B element () {
         throw new java.util.NoSuchElementException
            ("ReverseGadget.Empty.element()");
      }

      public FuncIterator<B> moveToNext () {
```

```
            throw new java.util.NoSuchElementException
               ("ReverseGadget.Empty.moveToNext()");
         }
      }

      private static class NonEmpty<B> extends ReverseGadget<B> {
         private B element;
         private FuncIterator<B> rest;

         public NonEmpty (B elt, FuncIterator<B> r) {
            element = elt;
            rest = r;
         }

         public boolean hasElement () {
            return true;
         }

         public B element () {
            return element;
         }

         public FuncIterator<B> moveToNext () {
            return rest;
         }
      }
}
```

Rather than explain the code, I will let you study it.

Here's an example to show that it does, in fact, reverse the result of an iterator:

```
show("Reverse(a)", ReverseGadget.create(a.funcIterator()));

show("Reverse(Cartesian(a,a))",
        ReverseGadget.create(CartesianGadget.create(a.funcIterator(),
                                                    a.funcIterator())));
```

with the expected output:

```
Reverse(a) =  3  2  1
Reverse(Cartesian(a,a)) =  (3,3)  (3,2)  (3,1)  (2,3)  (2,2)  (2,1)
  (1,3)  (1,2)  (1,1)
```