14 Code Reuse Example: Measurable Lists

Recall the List ADT, here specialized to work on integers if only to keep distractions at a minimum

```
CREATORS
  empty :      () -> List
  cons :       (Int, List) -> List

OPERATIONS
  isEmpty :    () -> Boolean
  first :      () -> Int
  rest :       () -> List
  append :     (List) -> List
  find :       (int) -> Boolean
```

Consider the implementation of lists via the Specification design pattern, where for reasons that will become clear we do *not* hide the implementation classes.

```
object List {

  def empty ():List = new ListEmpty()

  def cons (v:Int,l:List):List = new ListCons(v,l)
}


abstract class List {

  def isEmpty ():Boolean
  def first ():Int
  def rest ():List
  def find (f:Int):Boolean
  def append (M:List):List
}
```

```
class ListEmpty () extends List {

  def isEmpty ():Boolean = true
  def first ():Int = throw new RuntimeException("Empty.first()")
  def rest ():List = throw new RuntimeException("Empty.rest()")
  def find (f:Int):Boolean = false
  def append (M:List):List = M

  override def toString ():String = ""
}


class ListCons (v:Int,r:List) extends List {

  def isEmpty ():Boolean = false
  def first ():Int = v
  def rest ():List = r
  def find (f:Int):Boolean =
    (f==v) || r.find(f)
  def append (M:List):List = List.cons(v,r.append(M))

  override def toString ():String = v + " " + r
}
```

Suppose we want to extend that ADT with a `length()` and a `sum()` operation that compute the length and sum of a list, respectively. Clearly, we could add those operations to the implementation of lists, but suppose that we don't want to for whatever reason. (For instance, the code is part of a library and we don't have access to it, or can't modify it.)

So lets define a new ADT, the measurable lists ADT MLIST, and figure out the kind of code reuse we can apply to the situation to reuse the implementation of lists we already have. Here is the MLIST ADT:

```
CREATORS
  empty :     () -> MList
  cons :      (Int, MList) -> MList

OPERATIONS
  isEmpty :   () -> Boolean
  first :     () -> Int
  rest :      () -> MList
  length :    () -> Int
  append :    (MList) -> MList
```

```
find :      (Int) -> Boolean
length :    () -> Int
sum :       () -> Int
```

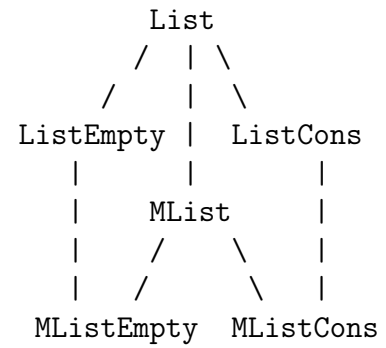
The specification for `length()` and `sum()` are the obvious ones. Also, it makes sense to have `MList` be a subtype of `List`.

Clearly, we could simply obtain an implementation of the MEASURABLE LIST ADT by applying the Specification design pattern. That's easy. (Do it if you're shaky on the basics.) But my goal now is to try to implement the MEASURABLE LIST ADT by reusing as much code as possible from the LIST ADT implementation.

Now, we could do our extension by simply defining a `length()` and `sum()` method in the subtype `Mlist` using the techniques we explicate below, but we can actually do better, that is, write a more efficient form of `length()` and `sum()`.

The naive implementation of `length()` which simply counts how many elements are in the list can be inefficient if the list is large. There is no way to make the "count how many elements are in the list" algorithm more efficient, but there is a way to implement measurable list to make the `length()` method more efficient: keep a count of the current list size alongside the list content, and simply increment the count upon a `cons()`. The `length()` method now simply returns the current list size, a constant-time field lookup operation. Something completely analogous can be done for `sum()`, by maintaining the current sum as we build the list. Dealing with lists in this way is an example of an *augmented data structure*, a data structure augmented with information that makes some operations more efficient.

So, how do we go about it? We could imagine that we want `MList` to inherit from `List`, but that does not actually make sense. Why? Because in our implementation of LIST obtained from the Specification design pattern, `List` is an abstract class, *and contains no method definitions.* There is nothing to inherit. Which makes inheritance useless, at least in so far as wanting to inherit from `List`. We can inherit from the implementation classes, though. Intuitively, we would like this subtyping picture:



And we would like to reuse the code from `ListEmpty` in `MListEmpty` and the code from `ListCons` in `MListCons`. So let's try that. First, though, let's do it delegation-style, because

it will reveal subtleties that will occur in the inheritance-based version of the code but in a more implicit form.

14.1 Delegation-based Implementation of MList

Our first implementation is based on delegation. We structure our code just like code obtained from the Specification design pattern, but we use delegation to implement the two implementation classes. First, the module and abstract class:

```
object MList {

  def empty ():MList = new MListEmpty()
  def cons (v:Int,l:MList):MList = new MListCons(v,l)
}


abstract class MList extends List {

  def isEmpty ():Boolean
  def first ():Int
  def rest ():MList
  def find (f:Int):Boolean
  def append (M:MList):MList

  def length ():Int
  def sum ():Int
}
```

The implementation classes are where the action is. Let's start with the `MListCons` implementation class. The idea behind delegation, remember, is that we can delegate the implementation of some of the methods to an implementation of an empty list that knows how to deal with those methods, and we can concentrate on the behavior that is specific for measurable lists.

```
class MListCons (v:Int,r:MList) extends MList {

  // delegate for cons list

  private val del:List = List.cons(v,r)   // we *know* r is an MList

  // some methods can be delegated to the cons list

  def isEmpty ():Boolean = del.isEmpty()
```

```
  def first ():Int = del.first()
  def find (f:Int):Boolean = del.find(f)
  override def toString ():String = del.toString()

  // some methods require a downcast from List to MList

  private def downcast (l:List):MList = l match {
    case l2:MList => l2
    case _ => throw new RuntimeException("Illegal downcast to MList")
  }

  def rest ():MList = downcast(del.rest())

  // other methods cannot actually be delegated (Exercise: why?)

  def append (M:MList):MList = MList.cons(v,r.append(M))

  // bridge methods

  def append (M:List):List = del.append(M)

  // actual functionality added by MListCons

  private val accLength:Int = 1+r.length()
  private val accSum:Int = v+r.sum()

  def length ():Int = accLength
  def sum ():Int = accSum
}
```

The delegate is defined using a field `del` initialized to a cons list created with the same argument, from the List ADT. Most of the methods, `isEmpty()`, `first()`, `find()`, and `toString()`, can simply delegate to that empty list.

Other methods are not so simple. Consider `rest()`. We can delegate to `del`, but the result returned from `del.rest()` is a `List`. We need `rest()` to return an `MList`. Now, an `Mlist` is a subtype of a `List`, so to go from a `List` to an `MList` we need to downcast, and the compiler will never downcast for us. So we need to write an explicit `downcast()` helper method, and use it to wrap the delegation of `rest()`.

To a first approximation, it seems that `append()` suffers from a similar problem: if we delegate to `del.append()`, then the result is a `List`, and we need to downcast to an `MList`. If we try the obvious definition:

```
  def append (M:MList):MList = downcast(del.append(M))
```

the type checker does not complain (why not?), but at run time the execution fails with a downcast exception throw from our `downcast()` function. The problem is that the call to `del.append()` results in a new `ListCons` instance being created, which cannot be downcast to an `MList` — and for good reason, since it is *not* an `MList`! So we cannot actually delegate `append()`, and need to implement it from scratch. There is no way to delegate the `append()` function.

The rest of the class is as expected. We need a bridge method for `append()` taking a `List` argument, which we can delegate to `ListCons`. And then we need the fields to record the length and sum of the list, capturing the actual functionality of measurable lists.

We do something exactly similar for the `MListEmpty` implementation class:

```
class MListEmpty () extends MList {

  // delegate for empty list

  private val del:List = List.empty()

  // some methods can be delegated to the empty list

  def isEmpty ():Boolean = del.isEmpty()
  def first ():Int = del.first()
  def find (f:Int):Boolean = del.find(f)
  override def toString ():String = del.toString()

  // some methods require a downcast from List to MList

  private def downcast (l:List):MList = l match {
    case l2:MList => l2
    case _ => throw new RuntimeException("Illegal downcast to MList")
  }

  def rest ():MList = downcast(del.rest())

  def append (M:MList):MList = M

  // bridge methods

  def append (M:List):List = del.append(M)

  // actual functionality added by MListEmpty
```

```
  private val accLength:Int = 0
  private val accSum:Int = 0

  def length ():Int = accLength
  def sum ():Int = accSum
}
```

And that's it. There is an alternative (simpler) implementation of the above that uses a common supertype to both `MListEmpty` and `MListCons` that captures the behavior that is in common between the two implementation classes, such as the methods that simply invoke the delegates, but I will leave that one as an exercise. The code above has the advantage of explaining what the inheritance-based implementation does below.

14.2 Inheritance-based Implementation of MList

Let's now implement the MList ADT using inheritance. We proceed as before, except now we want `MListEmpty` to be a subtype of both `MList` and `ListEmpty`, that latter to allow for inheritance of methods implemented in `ListEmpty`, and `MListCons` to be a subtype of both `MList` and `ListCons`. Because Scala lets us only extend one class, we need to make one of either `MList` and `ListEmpty` a trait. It makes sense to make `MList` a trait. Thus:

```
object MList {

  def empty ():MList = new MListEmpty()
  def cons (v:Int,l:MList):MList = new MListCons(v,l)
}


trait MList extends List {

  def isEmpty ():Boolean
  def first ():Int
  def rest ():MList
  def find (f:Int):Boolean
  def append (M:MList):MList

  def length ():Int
  def sum ():Int
}
```

What about the implementation classes? Let's start with `MListCons`. It extends `ListCons` (to allow for inheritance) and uses `with MList` to make it a subtype of `MList`. The thing about inheritance is that the system creates a delegate just like we did above, but that delegate is created *implicitly*. Whenever we create an instance of a class that `extends X`, the system creates a delegate for `X` inside the new instance. If a method is not defined in the instance of the class, then the system automatically delegates that method to the delegate. Because there is delegation going on, albeit implicit delegation, all the issues we highlighted above arise: some methods require downcasts, and some methods simply cannot be inherited. Here is the `MListCons` class using inheritance:

```
class MListCons (v:Int,r:MList) extends ListCons(v,r) with MList {

  private def downcast (l:List):MList = l match {
    case l2:MList => l2
    case _ => throw new RuntimeException("Illegal downcast to MList")
  }

  override def rest ():MList = downcast(super.rest())

  override def append (M:MList):MList = MList.cons(v,r.append(M))

  private val accLength:Int = 1+r.length()
  private val accSum:Int = v+r.sum()

  def length ():Int = accLength
  def sum ():Int = accSum
}
```

A few things to note. First, because the delegate is created implicitly, we need a way to specify the arguments to the delegate creation. (Recall, we need the delegate to have the same values as the instance we are creating.) We do so by specifying the arguments to the delegate after the supertype:

```
  extends ListCons(v,r)
```

This creates the implicit delegate as thought it was created using `new ListCons(v,r)`. Second, the methods that simply delegate need not be defined, since delegation will occur automatically for them. Third, for those methods where we need to access the result of the delegation to perform a downcast (e.g., `rest()`), we need a way to perform the delegation ourselves. The syntax for a delegate call when delegation is implicit is `super.m()` for some method `m()`.

We do something similar to `MListEmpty`:

```
class MListEmpty () extends ListEmpty with MList {

  private def downcast (l:List):MList = l match {
    case l2:MList => l2
    case _ => throw new RuntimeException("Illegal downcast to MList")
  }

  override def rest ():MList = downcast(super.rest())

  override def append (M:MList):MList = M

  private val accLength:Int = 0
  private val accSum:Int = 0

  def length ():Int = accLength
  def sum ():Int = accSum
}
```

So we have managed to reuse the code form `ListEmpty` and `ListCons` rather cleanly, but there is still a lot of common code between `MListEmpty` and `MListCons`. We could imagine pulling up the code in common into a supertype `Common` like we did last time, but note that this would require us to inherit from two classes at once, `ListCons` and `Common` in the case of `MListCons`. That's multiple inheritance, and we'll turn to that next time.