21 Abstracting Traversals

Introduction

We know by now that the same data may be represented in several different ways. We can save the information in a list, or in a binary search tree (BST), or maybe, there are other ways of keeping track of data. However, in each case we need to be able to perform the same basic operations. We can insert an item to a binary search tree, or add an item to a list, determine the number of elements in a tree or in a list, find the first item, or the structure that represents the rest of the items.

Another relevant observation is that over the time as we kept designing new methods for manipulating lists of data our classes kept growing and changing constantly. We had a choice of either carrying with us all the methods we already designed, or having several different variants of the same structures of data, each with methods relevant to our current problem.

Both of these problems suggest that we seek an abstraction. The first one suggests that the structure of the data and our way of interacting with this structure can be abstracted, so that we can interact in a uniform way with several different structures.

Abstracting over the structure of data

Suppose we defined an interface that allowed us to add items to a collection of data and to determine the number of elements in this collection. The interface would be:

```
// an interface to represent the construction of a data set
interface DataSet{

   // add the given object to this data set
   DataSet add(Object obj);

   // determine the number of elements in this data set
   int size();
}
```

It would be easy to modify our list structures and the binary tree structure to implement this interface. For the lists, we do the following:

```
// in the class ALoObj:

  // add the given object to this data set
  DataSet add(Object obj){
    return new ConsLoObj(obj, this);
  }

  // determine the number of elements in this data set
  abstract int size();

  // in the class MTLoObj:
  int size(){
    return 0;
  }

  // in the class ConsLoObj:
  int size(){
    return 1 + this.rest.size();
  }
```

though we may think of a better way of dealing with the size. For the BST the *add* method would be the same as the original *insert* and again, the size is the same as the count of nodes we designed earlier. However, now we can build either of these structures using the same methods in the same way.

We will see more of these kinds of abstractions when we discuss the *Java Collections Framework*.

Abstracting over the traversals

The second problem is a bit harder. Each method that processed a list of data engaged every element of the list in some computation, one element at a time. We may have been just counting them, of selecting those that satisfied some predicate, or producing a new value from the data contained in the original data element (map).

The typical structure of the program was:

```
// in the class ALoObj:
abstract Object method(...);
```

```
// in the class MTLoObj:
Object method(...){
   return baseValue;
}

// in the class ConsLoObj:
Object method(...){
   return result using
                  ... (this.first) ...
                  ... this.rest.method(...)...);
}
```

We looked at two specific methods:

```
// determine the number of elements in this data structure
int size();

// does this list contain the given object?
boolean contains(Object obj);
```

The method bodies in the two classes, *MTLoObj* and *ConsLoObj* were:

```
// in the class MTLoObj:
int size() {
   return 0;
}

boolean contains(Object obj){ return false;
}

// in the class ConsLoObj:
int size(){
   return 1 + this.rest.size();
}

boolean contains(Object obj){
   return ((ISame)this.first).same(obj) ||
          this.rest.contains(obj);
}
```

Not all parts were present for all problems, but it is clear that we needed to be able to process the first item and to have access to the rest. Let us recall similar methods in Scheme. The general structure of a function defined for a list-like data was:

```
(define (fcn alist)
  (cond
    [(empty? alist) ... produce base-value ...]
    [(cons? alist)  ... produce result using
                                        ... (first alist) ...
                                        ... (fcn (rest alist)) ...]))
```

We would like to be able to design a method in our *Examples* class that consumes a list structure and produces a result that is computed by examining each element of that list. We need to design a mechanism that will allow us to access the data in the list in the desired orderly fashion.

We start by defining the desired interface that allows us to observe the contents of a list-like structure:

```
// functional iterator for a linear traversal of a data structure
interface Traversal{

  // is there a current element available in the structure
  boolean hasMore();

  // produce the current element of the structure
  Object current();

  // produce an iterator for the rest of this structure
  Traversal advance();
}
```

We now need to implement these methods in the classes that represent a list of objects. the method *hasMore* produces false for the empty list and produces true for the nonempty list. The methods *current* and *advance* cannot produce any sensible result for the empty list and should signal an error. The *current* element in a nonempty list is the value off the first field. The list we get by advancing beyond the first element is the rest. So, the complete implementation of the *Traversal* interface is as follows:

```
interface ILoObject extends Traversal{

  // is there a current element available in the structure
  boolean hasMore();

  // produce the current element of the structure
  Object current();
```

```
    // produce an iterator for the rest of this structure
    Traversal advance();
}

class MTLoObject implements ILoObject{
    MTLoObject() {
}

    // is there a current element available in the structure
    boolean hasMore(){
        return false;
    }

    // produce the current element of the structure
    Object current(){
        throw new NoSuchElementException(
            "Cannot produce current element in an empty list");
    }

    // produce an iterator for the rest of this structure
    Traversal advance(){
        throw new NoSuchElementException(
            "Cannot advance in an empty list");
    }
}

class ConsLoObject implements ILoObject{
    Object first;
    ILoObject rest;

    ConsLoObject(Object first, ILoObject rest){
        this.first = first;
        this.rest = rest;
    }

    // is there a current element available in the structure
    boolean hasMore(){
        return true;
    }
```

```
    // produce the current element of the structure
    Object current(){
      return this.first;
    }

    // produce an iterator for the rest of this structure
    Traversal advance(){
      return this.rest;
    }
  }
```

We can now design the two methods, *size* and *contains*. Recall their earlier definitions:

```
  interface ILoObject{
    // to compute the size of this list
    int size();

    // is the given book in this list?
    boolean contains (Object that);
  }

  class MTLoObject implements ILoObject{
    MTLoObject() {}

    int size(){
      return 0;
    }

    boolean contains (Object that){
      return false;
    }
  }

  class ConsLoObject implements ILoObject{
    Object first;
    ILoObject rest;

    ConsLoObject(Object first, ILoObject rest){
      this.first = first;
      this.rest = rest;
    }
```

```
    int size(){
      return 1 + this.rest.size();
    }

    boolean contains (Object that){
      return ((ISame)this.first).same(that)
             || this.rest.contains(that);
    }
}
```

The translation of these methods into *external* methods

```
int size(Traversal t);

boolean contains(Object obj, Traversal t);
```

defined in the *Examples* class or some other client class is straightforward. We first write down the template for the method that consumes the *Traversal* iterator. Note that the value of **this** is irrelevant here — the instance of the *Examples* class is not being used by the method.

```
... t ...
... t.hasMore() ...
    ... if ((t.hasMore())
            ... t.current() ...
            ... t.advance() ...
            ... size(t.advance()0 ...
            ... contains(Object, t.advance()) ...
```

The complete implementation of these methods then becomes:

```
// compute the size of the data set given by the traversal
int size(Traversal t){
  if (t.hasMore())
    return 1 +
           size(t.advance());
  else
    return 0;
}
```

```
// does the data set given by the traversal contain the given object?
boolean contains(Object obj, Traversal t){
  if (t.hasMore())
    return ((ISame)t.current()).same(obj) ||
           contains(obj, t.advance());
  else
    return false;
}
```

We just have to convert earlier examples into tests using the new methods. The original examples for the lists of books were:

```
class Examples{
  Examples () {
  }

  Book b1 = new Book("DVC", 2003);
  Book b2 = new Book("LPP", 1942);
  Book b3 = new Book("HtDP", 2001);

  ILoObject mtbooks =
           new MTLoObject();
  ILoObject booklist =
           new ConsLoObject(b1,
           new ConsLoObject(b2,
                            mtbooks));

  boolean testSizeBook1 =
    mtbooks.size() == 0;
  boolean testSizeBook2 =
    booklist.size() == 2;

  boolean testContainsBook1 =
    this.mtbooks.contains(this.b1) ==
      false;
  boolean testContainsBook2 =
    this.booklist.contains(this.b2) ==
      true;
  boolean testContainsBook3 =
    this.booklist.contains(b3) ==
      false;
}
```

The new tests — after the methods *size* and *contains* are defined in the *Examples* class are:

```
boolean testSizeBookT1 =
  this.size(this.mtbooks) == 0;
boolean testSizeBookT2 =
  this.size(this.booklist) == 2;

boolean testContainsT1 =
  this.contains(this.b1, this.mtbooks) ==
    false;
boolean testContainsT2 =
  this.contains(this.b2, this.booklist) ==
    true;
boolean testContainsT3 =
  this.contains(b3, this.booklist) ==
    false;
```