Data Structures, ADTs, and PDAs

Based on notes by Logan Mayfield

In these notes we explore the paradigm of designing Procedural Data Abstractionsand defining recursively structured data. In their current state these notes are woefully incomplete. For a more complete treatment of basic recursive structures see chapters 5 and 15 in HtDC[3]. For a deeper look at abstractions in this context look at chapters 18 and 19.

Lists and Procedural Data Abstractions

Lists are the classic example of a data structure with a recursive structure. A list comes in two varieties: empty and not empty. The empty list is a primitive structure with no contained fields/data. A non-empty list, which we'll call a cons list borrowing from the Lisp/Scheme tradition, has two fields. The first field is a singular instance of the type contained in a list.[1] The second field is a pointer/reference to another list. It is this second field that creates a recursive structure: cons lists are composed of a single datum and another list.

[1] For lists of integers, it's an integer, for a list of Strings, it's strings.

In an object-oriented space we can represent this structure using a class union and containment. Nothing new is happening, we're just making use of existing tools in a new and highly fruitful way. The raw structure and a standard interface for a list of Foo objects is given in figure 1. Although it might not be clear why just yet adding a mutable interface to Lists isn't possible with this design[2].

[2] what happens when you remove the only item from a size one list?

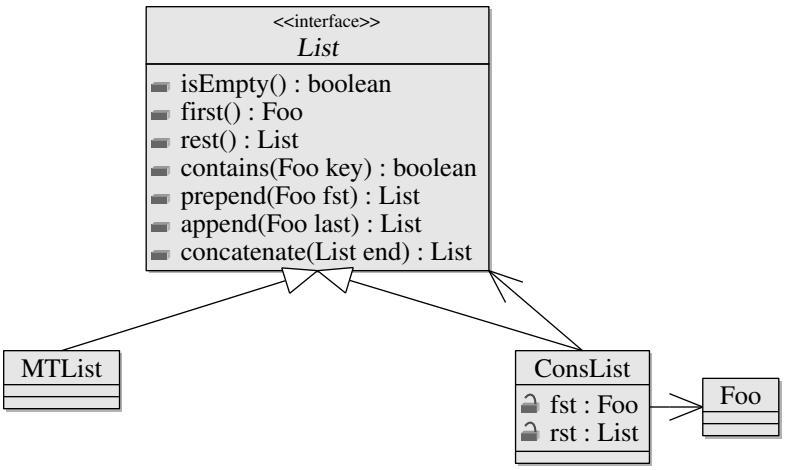

Figure 1: A List of Foo Objects

In practice, the Foo class could be a primitive value, an interface-based type, or any other more concrete/descriptive type than "Foo". By embedding the recursive structure in containment and inheritance we can get the system to manage the necessary "is this list empty or

not" conditionals as part of the Polymorphic method dispatch. Implementing the interface across this union is an exercise in Procedural Data Abstraction[1, 2].

The *first* method, as seen in figure 2, is very straight forward for cons lists as it's really just a field selector.

Figure 2: first for cons lists

```
// in ConsList
@Override
public Foo first(){
  return this.fst;
}

// in ConsList test file
@Test
public void testFirst(){
  assertEquals(new Foo(),new ConsList(new Foo(), new MtList()));
}
```

On the other hand, figure 3 shows that first for empty lists is an error. There is no first to select when there is nothing in the list. Using a generic RuntimeException is not ideal, but works. In practice you should probably extend RuntimeExecption to create a custom exception type for List errors. When testing exceptions we need to add an argument to the annotation and then simply invoke code that should throw the exception listed in the annotation.

Figure 3: first for empty lists

```
// in MtList
@Override
public Foo first(){
  throw new RuntimeException("Cannot select the first of an empty
      list");
}

@Test
public void testFirst() {
 assertThrows(RuntimeException.class, () -> { new
     MtList().first(); }, "message");
}
```

The prepend function is also fairly straight forward.

Concatenate is, at first, a bit tricky as we must be certain we account for the two possible List subclasses that *end* could be. The problem, in general, has four logical cases: both are empty, this is empty while end is not, end is empty while this is not, and neither is empty. While polymorphic method dispatch fully determines the type of *this*, it does nothing relative to the argument types.[3] It's up to us to manage the two sub-cases relative to the type of this.

In the case of concatenate, we find that the logic when end is

[3] some OOP languages will dispatch based off argument types. this is called Multiple Dispatch.

```
// in ConsList
@Override
public List prepend(Foo fst){
  return new ConsList(fst,this);
}

// in ConsList test file
@Test
public void testPrepend(){

  List l = new ConsList(new Foo(b),new MTList());
  assertEquals(new ConsList(new Foo(a),new ConsList(new Foo(b),new
      MTList())),
    l.prepend(new Foo(a));
}
```

Figure 4: prepend for cons lists

```
// in MtList
@Override
public List prepend(){
  return new Conslist(fst,this);
}

// in MtList test file
@Test
public void testPrepend(){
  List l = new ConsList(new Foo(b),new MTList());
  assertEquals(new ConsList(new Foo(a),new MTList()),
    new MTList().prepend(new Foo(a)));
}
```

Figure 5: prepend for empty lists

empty and when it's not empty can be unified into a single case for both non-empty and empty this. Figure 6 shows the implementation for cons lists and figure 7 shows it for empty lists.

Figure 6: concatenate for cons lists

```
// in ConsList
@Override
public List concatenate(List end){

  return new ConsList(this.first(), this.rest().concatenate(end));

}

// in ConsList test file
@Test
public void testConcat(){
  List l = new ConsList(new Foo(a),new ConsList(new Foo(c),new
      MTList()));
  List r = new ConsList(new Foo(b),new MTList());

  assertEquals(new ConsList(new Foo(a),
           new ConsList(new Foo(c),
             new ConsList(new Foo(b),
               new MTList()))),
       l.concatenate(r));

  assertEquals(l,l.concatenate(new MTList());

}
```

ADT Lists

The recursive structure given above is really a low level implementation choice. If you need a list or list like container you should be working with an Abstract Data Type and then using the OO list structure as the implementation. In figure 8 we see the structure for a basic ADT list with a linked-list implementation. Notice that the list methods move up to the ADTList class. Now that the List type is implementation we no longer are required to provide the standard list interface. It would make a lot of sense to do so and have ADTList methods make calls to the logically equivalent method on the contained List. On the other hand, we might opt to check the specific type of the list to catch special cases in the List logic. For example, we could skip the call to List concatenate all together if the argument to the ADTList concatenate is empty. Once again, the design space between the implementation and the top-level has opened up some implementation flexibility.

This works perfectly well when you're certain that the linked-list implementation provides the performance characteristics you need. In the event that these characteristics are unknown or that you simply

Figure 7: concatenate for empty lists

```
// in MtList
@Override
public List concatentate(List end){
  return end;
}

// in MTList test file
@Test
public void testConcat(){
  assertEquals(new MTList(), new MTList().concatenate(new
      MTList()));

  assertEquals(new ConsList(new Foo(),new MTList()),
    new MTList().concatenate(new ConsList(new Foo(), new
        MTList())));

}
```

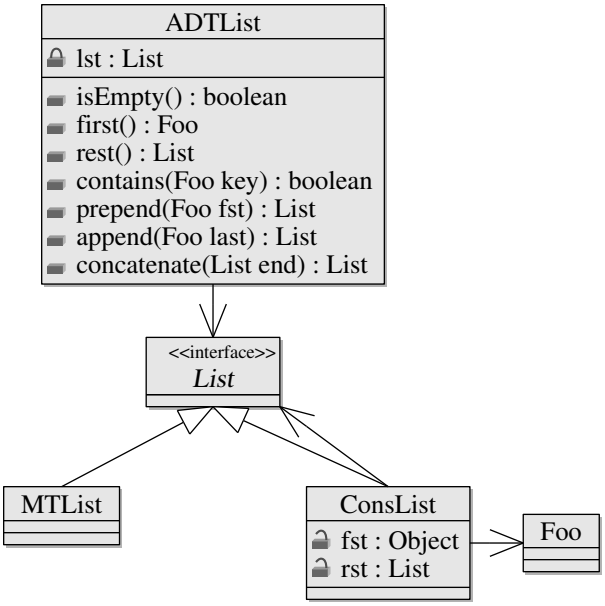

Figure 8: An ADT List of Foo Objects

want to plan for a more flexible future then we want to inject some interfaces into this picture. In figure 9 we see a more robust ADT list structure that defines the main type through an interface and then sets down two possible implementations: one with an array and one with a linked-list.

The key observation to make about figure 9 is that *different implementations form a class Union*.

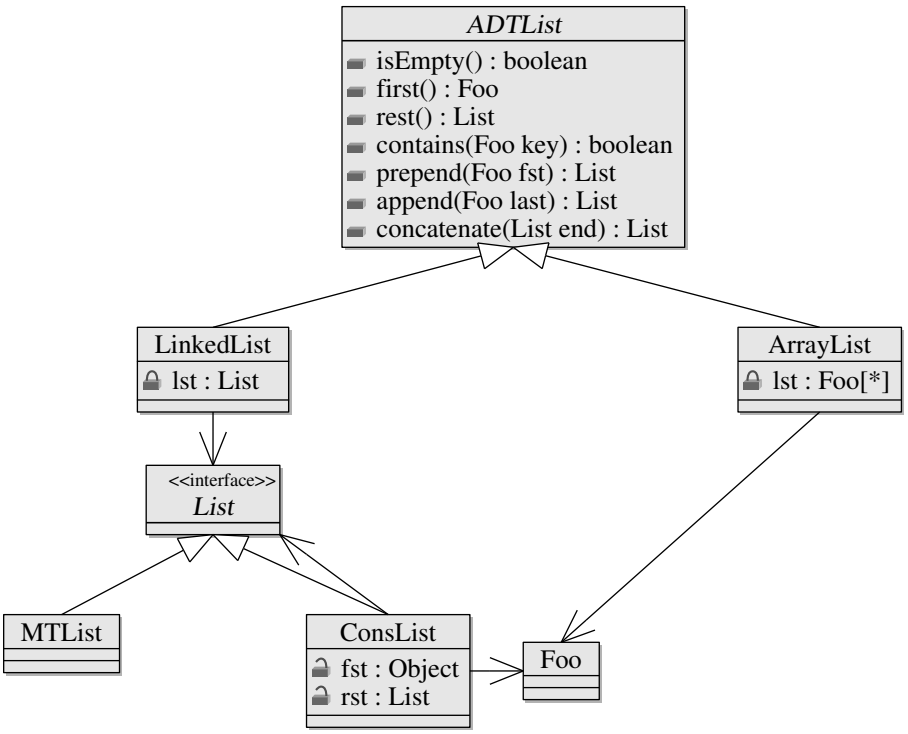

Figure 9: An ADT List of Foo Objects

References

[1] William R. Cook. Object-oriented programming versus abstract data types. In *Proceedings of the REX School/Workshop on Foundations of Object-Oriented Languages*, pages 151–178, London, UK, UK, 1991. Springer-Verlag.

[2] William R. Cook. On understanding data abstraction, revisited. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, pages 557–572, New York, NY, USA, 2009. ACM.

[3] Matthias Felleisen, Matthew Flatt, Robert Bruce Findler, Kathryn E. Gray, Shriram Kirshnamurthi, and Viera K. Proulx. How to design classes. http://www.ccs.neu.edu/home/matthias/htdc.html, 6 2012.