COMP 210 - Lecture Notes - 07 - Data Structures, ADTs, and PDAs

February 9, 2016

In these notes we explore the paradigm of designing Procedural Data Abstractions[1] and 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[2]. For a deeper look at abstractions in this context look at chapters 18 and 19.

[1] William R. Cook. 1990. Object-Oriented Programming Versus Abstract Data Types. In Proceedings of the REX School/Workshop on Foundations of Object-Oriented Languages, J. W. de Bakker, Willem P. de Roever, and Grzegorz Rozenberg (Eds.). Springer-Verlag, London, UK, UK, 151-178.

[2] 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

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.[3] 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.

[3] 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 of a list of Foo objects is given in figure 1. In practice, the Foo class could be a primitive value, an interface-based type, or any other more concrete/descriptive type than "Foo".

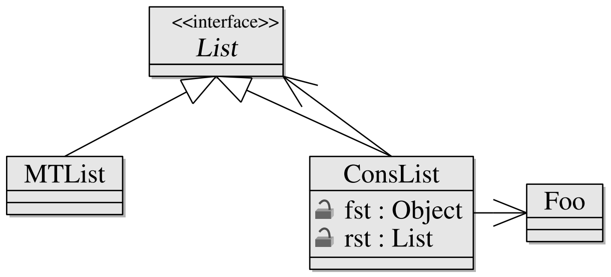

Figure 1: A List of Foo Objects

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.

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 2 we see the structure for a basic ADT list with a linked-list implementation.

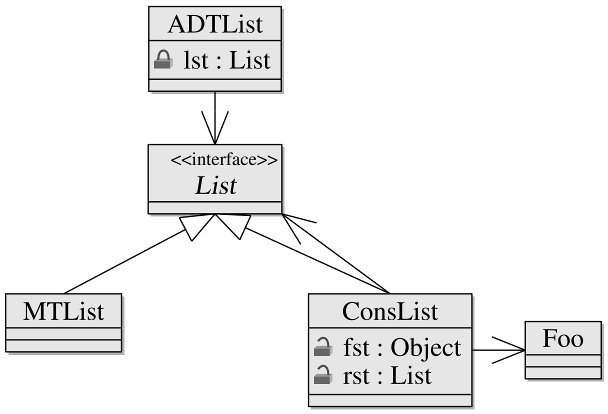

Figure 2: An ADT List of Foo Objects

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 want to plan for a more flexible future then we want to inject some interfaces into this picture. In figure 3 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.

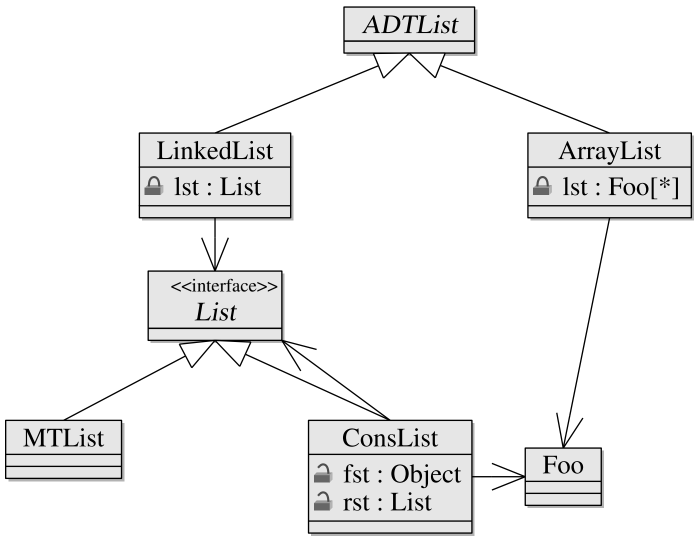

Figure 3: An ADT List of Foo Objects

The key observation to make about figure 3 is that *different implementations form a class Union*.

References

[1] 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.