OOP as an Enrichment of FP

Robert "Corky" Cartwright
Rice University

Abstract

At Rice University, we have taught functional programming (FP) in Scheme in our first semester programming course for the past two decades. Over ten years ago, when we converted our second semester course from object-based programming (OBP) in C++ to object-oriented programming (OOP) in Java, we discovered that object-oriented program design and functional programming design are intimately connected. In both FP and OOP:

1. Most program data is *algebraic*; it has a simple inductive definition as trees.

2. The structure of a program should mirror the form of data that it processes.

Moreover, for every functional program, there is a corresponding object-oriented program derived from the functional program using appropriate design patterns. The resulting OO program is still functional in the sense that all data objects are immutable.

Since the OO programming model is more complex that the functional one, we advocate teaching the rudiments of functional programming *before* teaching object-oriented design. Moreover, OO design pedagogy should initially focus on programming with immutable data using the design patterns that elegantly encode standard programming techniques from functional programming. The patterns include composite, interpreter, singleton, strategy, factory method, and visitor patterns. Students should learn that there is an OO analog for every abstraction in FP. In Java and C#, the connection is particularly compelling because Java and C# supports closures in the form of anonymous inner classes and anonymous delegates, respectively. Once students have mastered "functional programming" in Java or C#, it is natural to introduce object mutation using the state, iterator, observer, and model-view-controller patterns.

A pedagogic IDE such as DrJava (see drjava.org) can greatly simplify the complexity of writing "functional" OO programs by autogenerating the methods implied in the definition of algebraic data, just as the `define-struct` operation in Scheme autogenerates all of the operations (constructors, selectors, and structural equality) for Scheme structures. The autogeneration process is natural because the generated code exactly matches what a competent OO programmer would produce to fully implement an algebraic data type. In fact, we believe that professional IDEs should offer similar support. As students learn more about the full Java or C# language, they easily learn to write the autogenerated code on their own (although this aspect of Java and C# adds to clerical burden involved in writing "functional" OO programs.

OOP as an Enrichment of FP

Corky Cartwright

WG 2.8
19 Jun 2008

References

- R. Cartwright, D. Nguyen. FP as an Enrichment of OOP, In *OOPSLA 2001 Workshop on Pedagogies and Tools for Assimilating Object-Oriented Concepts*
  http://www.cs.umu.se/~jubo/Meetings/OOPSLA01/Contributions/RCartwright.html
- J. Hsia, E. Simpson, D. Smith, R. Cartwright. Taming Java for the Clasroom. In SIGCSE 2005, available at drjava.org.
- Comp 211 course modules, Rice University, at the *Connexstion Project* site, www.cnx.org, (still in preparation)
- TeachJava notes. Available at
  http://www.cs.rice.edu/~cork/teachjava/2002/notes/current.pdf
- Java design recipe. Available at
  http://www.cs.rice.edu/~cork/teachjava/2002/NewRecipe.html

Central Thesis

- Data-directed program design as explained in *How to Design Progams* (by Felleisen, Findler, Flatt, & Krishnamurthi) is directly applicable to Java given appropriate support from a pedagogic IDE such as DrJava.
- Programming process guided by a design recipe
- Inductively defined data domains
- Program operations written using a template expressing structural recursion on the form of data being processed.
- Programs are tested using input/output examples written before the code (test-driven development).

FP -> OOP

- Inductive data definition -> Composite pattern with same variants as data definition

  <parentType> ::= variant1 | ... | variantN

  abstract class <parentType> {}
  class variant1 extends parentType { ... }
  ...
  class variantN extends parenType { ... }

- Natural recursion template -> interpreter pattern with recursive calls expected on fields of recursive type