1 Object-oriented programming vs. functional programming

We’ve now seen the core features of both OOP and FP. Let’s compare the two paradigms, and in so doing, reach a better understanding of both.

As a running example, let’s consider implementing a small programming language (not coincidentally, this is a problem we have already considered a couple times in the course). Suppose we have expressions and operations we want to compute over expressions. Naturally, we’ll have

- different variants of expressions—for example, integer values, negation expressions, and addition expressions—and
- different operations over expressions—for example, evaluating them, converting them to strings, or determining if they contain the constant zero in them.

This problem leads to a conceptual matrix (two-dimensional grid) with one entry for each combination of variant and operation:

<table>
<thead>
<tr>
<th></th>
<th>eval</th>
<th>toString</th>
<th>hasZero</th>
</tr>
</thead>
<tbody>
<tr>
<td>Int</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Add</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Negate</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

No matter what programming language you use (Ruby, SML, etc.), you have to write code to describe the computation performed for each entry in the matrix. So let’s look at how that code would be written in a functional language vs. an OO language.

2 The functional approach

The usual idiom for writing code to implement a little language is as follows:

- Define a single datatype for expressions. That datatype has multiple constructors, one for each variant.
- Define multiple functions, one for each operation.
- In each function, write a branch for each variant. (In SML, you might use a case expression and pattern matching. If there is a default for many variants, we can use something like a wildcard pattern to avoid enumerating all the branches.)

This approach primarily focuses on procedural decomposition: breaking the problem down into procedures corresponding to each operation.

The following ML code demonstrates this approach. Notice how we define all the kinds of expressions in one place, and how the nine entries in the matrix are implemented “by column” (with one function for each column):

```
exception BadResult of string
```
datatype exp =
    Int of int
  | Negate of exp
  | Add of exp * exp

fun eval e =
  case e of
    Int _ => e
  | Negate e1 => (case eval e1 of
      Int i => Int (~i)
    | _ => raise BadResult "non-int in negation")
  | Add(e1,e2) => (case (eval e1, eval e2) of
      (Int i, Int j) => Int (i+j)
    | _ => raise BadResult "non-ints in addition")

fun toString e =
  case e of
    Int i => Int.toString i
  | Negate e1 => "-(" ^ (toString e1) ^ ")"
  | Add(e1,e2) => "(" ^ (toString e1) ^ " + " ^ (toString e2) ^ ")"

fun hasZero e =
  case e of
    Int i => i=0
  | Negate e1 => hasZero e1
  | Add(e1,e2) => (hasZero e1) orelse (hasZero e2)

3 The object-oriented approach

In object-oriented languages, the usual idiom for implementing little languages is as follows:

• Define a single class for expressions, with one abstract method for each operation.
• Define multiple subclasses of the expression class, one for each variant.
• In each subclass, define multiple methods, one for each operation. (If there is a default for many variants, we can define it in the superclass and inherit that default behavior.)

This approach primarily focuses on data-oriented decomposition: breaking the problem down into data structures corresponding to each variant.

The following Java code demonstrates this approach. Notice how we define all the operations in one place (the abstract class), and how the the nine entries in the matrix are implemented “by row” (with one class for each row):

abstract class Exp {
    abstract Value eval();
    abstract String toString();
    abstract boolean hasZero();
}
class Int extends Exp {
    public int i;

1 We use the name toString instead of toString to avoid overriding the toString method in class Object.
Int(int i) {  
    this.i = i;  
}  
Value eval() {  
    return this;  
}  
String toString() {  
    return "" + i;  
}  
boolean hasZero() {  
    return i==0;  
}

class Negate extends Exp {  
    public Exp e;  
    Negate(Exp e) {  
        this.e = e;  
    }  
    Value eval() {  
        // we downcast from Exp to Int, which will raise a run-time error  
        // if the subexpression does not evaluate to an Int  
        return new Int(- ((Int)(e.eval())).i);  
    }  
    String toString() {  
        return "-(" + e.toString() + ")";  
    }  
    boolean hasZero() {  
        return e.hasZero();  
    }  
}

class Add extends Exp {  
    Exp e1;  
    Exp e2;  
    Add(Exp e1, Exp e2) {  
        this.e1 = e1;  
        this.e2 = e2;  
    }  
    Value eval() {  
        // we downcast from Exp to Int, which will raise a run-time error  
        // if either subexpression does not evaluate to an Int  
        return new Int(((Int)(e1.eval())).i + ((Int)(e2.eval())).i);  
    }  
    String toString() {  
        return "(" + e1.toString() + " + " + e2.toString() + ")";  
    }  
    boolean hasZero() {  
        return e1.hasZero() || e2.hasZero();  
    }  
}

The following Ruby code is similar. Note how we don’t need to define abstract methods or insert type casts in Ruby. (Even the superclass is unnecessary, but we leave it in for clarity.)
class Exp
  # could put default implementations or helper methods here
end

class Int < Exp
  attr_reader :i
  def initialize i
    @i = i
  end
  def eval
    self
  end
  def toString
    @i.to_s
  end
  def hasZero
    i==0
  end
end

class Negate < Exp
  attr_reader :e
  def initialize e
    @e = e
  end
  def eval
    Int.new(-e.eval.i) # error if e.eval has no i method (not an Int)
  end
  def toString
    "-(" + e.toString + ")"
  end
  def hasZero
    e.hasZero
  end
end

class Add < Exp
  attr_reader :e1, :e2
  def initialize(e1,e2)
    @e1 = e1
    @e2 = e2
  end
  def eval
    Int.new(e1.eval.i + e2.eval.i) # error if e1.eval or e2.eval have no i method
  end
  def toString
    "(" + e1.toString + " + " + e2.toString + ")"
  end
  def hasZero
    e1.hasZero || e2.hasZero
  end
end

4 Comparison #1

As we've just seen,
• functional decomposition defines computations (i.e., programs) in terms of functions. Those functions are in turn defined in terms of data structures (i.e., datatypes). Whereas,

• object-oriented decomposition defines computations (i.e., programs) in terms of data structures (i.e., classes). Those data structures are in turn defined in terms of functions (i.e., methods).

These are exactly opposite ways of defining the same computation. In fact, both are ways of defining the matrix we started with—the only difference is whether to lay out a program “by column” (the FP approach) or “by row” (the OO approach).

Understanding this symmetry is invaluable deciding how to decompose a problem. Moreover, IDEs can help you view a program in a different way than the source code is decomposed. For example, Java code is organized “by row”, but in Eclipse by selecting a method name and showing the quick type hierarchy, you can get a kind of “by column” view of the code.

So, is one approach better than the other? It’s often a matter of personal preference whether it seems “more natural” to lay out a program by row or by column, so you are entitled to your opinion. Experience shows that, for some kinds of programs, one approach tends to work better than the other. For our little expression language, the functional approach might be more popular: it is “more natural” to have the cases for eval together rather than the operations for Negate together. For implementing GUIs, the object-oriented approach is probably more popular: it is “more natural” to have the operations for a kind of data (like a MenuBar) together rather than have the cases for doIfMouseIsClicked together. The choice can also depend on what programming language you are using, how standard libraries are organized, etc.

However, there are times when the difference between the two approaches is decidedly not subjective, and there is clearly an answer as to which is better...

5 Extending the language

Suppose we now need to extend our little expression language with new kinds of expressions (e.g., multiplication expressions) and new operations (e.g., transforming an expression so that it has no negative constants in it).

The functional approach. Adding a new operation is easy: we can implement a new function without modifying any existing code. For example, this function creates a new expression that evaluates to the same result as its argument but has no negative constants:

```plaintext
fun noNegConstants e =
  case e of
    Int i => if i < 0 then Negate (Int(~i)) else e
  | Negate e1 => Negate(noNegConstants e1)
  | Add(e1,e2) => Add(noNegConstants e1, noNegConstants e2)
```

But adding a new kind of expression, such as Mult of exp * exp, is not easy. We need to go back and edit all of our operations to add a new branch to the pattern match. In a statically-typed language like SML, we do get some help from the compiler: after adding the Mult constructor, if our original code did not use wildcard patterns, then the type-checker will give a non-exhaustive pattern-match warning everywhere we need to add a case for Mult.

The object-oriented approach. Adding a new expression is easy: we can implement a new subclass of Exp without editing any existing code. For example, this Java class adds multiplication expressions to our language:
class Mult extends Exp {
  Exp e1;
  Exp e2;
  Mult(Exp e1, Exp e2) {
    this.e1 = e1;
    this.e2 = e2;
  }
  Value eval() {
    return new Int(((Int)(e1.eval())).i * ((Int)(e2.eval())).i);
  }
  String toString() {
    return "(" + e1.toString() + " * " + e2.toString() + ");";
  }
  boolean hasZero() {
    return e1.hasZero() || e2.hasZero();
  }
}

But adding a new operation, such as noNegConstants, is not easy. We have to go back and edit every subclass of Exp to add a new method. In a statically-typed language, we do get some help from the compiler: after adding abstract method abstract Exp noNegConstants(); to the Exp class, the type-checker will give an error for any non-abstract class that needs to implement the method.

6 Comparison #2

As in Comparison #1, FP and OOP turn out to be exact opposites. It’s easy to add new columns to the matrix in FP, but hard to add new rows. OOP makes it easy to add new rows, but hard to add new columns.

So if we know in advance that, as part of ongoing maintenance of our code, we’ll need to add new data variants frequently, we might be wise to choose to implement our code in an OOP language. Likewise, if we know we’ll need to add new operations frequently, we could choose an FP language.

This comparison is not subjective. Rather, the programming language itself makes certain kinds of extension easier, and other kinds harder.

7 Planning for extensibility

It’s possible for functional decomposition to support new variants, or for object-oriented decomposition to support new operations, if you plan ahead and use somewhat awkward programming techniques.

For object-oriented programming, the visitor pattern is a common approach. We won’t discuss that technique here.

For functional programming, we can define our datatypes to have an “other” possibility and our operations to take in a function that can process the “other” data. Here is the idea in SML:

datatype 'a ext_exp =
  Int of int
  | Negate of 'a ext_exp
  | Add of 'a ext_exp * 'a ext_exp
  | OtherExtExp of 'a

  fun eval_ext (f,e) = (* notice we pass a function to handle extensions *)
case e of
  Int i      => i
| Negate e1 => 0 - (eval_ext (f,e1))
| Add(e1,e2) => (eval_ext (f,e1)) + (eval_ext (f,e2))
| OtherExtExp e => f e

With this approach, we could create an extension supporting multiplication by instantiating ‘a with exp * exp, passing eval_ext the function (fn (x,y) => eval_ext(f,e1) * eval_ext(f,e2)), and using OtherExtExp(e1,e2) for multiplying e1 and e2. This approach can support different extensions, but it does not support combining two separately created extensions.

Notice that it does not work to wrap the original datatype in a new datatype like this:

datatype myexp_wrong =
  OldExp of exp
| MyMult of myexp_wrong * myexp_wrong

That approach does not allow, for example, a subexpression of an Add to be a MyMult.

**Final thoughts on extensibility.** The future is often difficult to predict; we might not know what kinds of extensions are likely to be needed, hence we might not be able to make an informed choice about OOP vs. FP. Moreover, both forms of extension might be likely. Newer languages like Scala aim to support both forms of extension well; we’re still gaining practical experience on how well it works.

Building software that is both robust and extensible is difficult. Extensibility can make the original code more work to develop, harder to reason about locally, and harder to change without breaking extensions. In fact, languages often provide constructs exactly to prevent extensibility. For example, Java’s final modifier on a class prevents subclasses.