Does Functional Programming Really Matter?

Two Approaches to Comparing
Functional and Object-Oriented Programming

Arvind Narayanan
Paul A Hansen
Programming Languages (CS 386L)

1. Introduction

To answer the question of whether or not the code examples in "Why Functional Programming Matters" truly constitute a demonstration of the superiority of functional programming, one must first ask what it means for those programs to be implementable in other languages. Most languages of interest are Turing complete, and therefore compute the same functions. Clearly, then, we are looking for a more direct transformation into the target language.

One approach would be to consider the features of functional programming that are claimed by Hughes to set it apart, and then to see if each one of these can be *emulated* in a simple manner in the target language. If we fail to emulate a feature, we gain a good understanding of where the language falls down; if we succeed, we can go ahead and use an empirical method as added evidence that the target language is as powerful as Lisp.

Another alternative would be to look at how programs in the source language might be implemented in the target language, in effect creating a *translation* from one language to another. This can be accomplished either by re-implementing the examples directly, or by examining correspondences between the languages in a more general fashion.

In this project, we demonstrate a Java emulation (section 2) and a C# translation (section 3) of the Hughes code snippets. We conclude that (modulo a few tradeoffs) the same level of modularity and code reuse can be achieved in object-oriented programming as in functional programming, making it more a choice of style than of necessity. We also present some benefits of object-oriented programming from the software engineering perspective. We discuss these tradeoffs (section 4).

2. Emulation

In the emulation approach, our goal is to create a layer of "wrappers" over standard Java classes so that Lisp code can be converted more or less automatically into Java while still keeping in the spirit of object-oriented programming. This is a tricky balance to achieve. But first, let us discuss what it means to emulate a feature of one language in another, in the light of Turing completeness of the target language. We will start with some examples of emulation to gain a better understanding.

Some features are pure syntactic sugar: i.e. a `do...while` loop can be trivially emulated in a language with a `while` loop. At the other extreme are features like the `eval` function in Lisp (or Python), which executes the code passed to it as argument. To emulate this feature in a language that does not have it, such as C or Java, one would have to embed an interpreter or a copy of the compiler into the compiled program. It does not require further argument to assert that the `eval` feature is nontrivially powerful.

Higher order functions. Hughes puts forth higher order functions and lazy evaluation as the unique features of Lisp. How hard is it to emulate higher order functions in C? It can be done by completely short-circuiting the type system entirely, i.e. by typing all function arguments and return types as `void *`. To regain type checking, it is necessary to embed a type checker into the compiled program. Thus the effort involved here is less than in the previous example (duplicating the type checker vs. duplicating the compiler), but nonetheless intolerably high.

In Java, on the other hand, the situation is very different, because of the availability of subclassing and inheritance. To emulate higher order functions, we bypass static type checking except for the arity of functions. Having to specify function arity is a minor inconvenience, the alternative being the use of variable argument lists available in Java 1.5. On the other hand, putting functions of different arity in different classes makes the code clearer. A Lisp function is modeled by default as an interface with a single method, but functions may be grouped when called for. Again, there is a tradeoff between the modularity offered by the class abstraction and the need to wrap even isolated methods by class definitions. Different classes implementing the interface provide completely different implementations for the same method, emulating the ability to take a function as argument. Downcasting provides the run-time type checking that is needed in order to have meaningful typing.

We have only a small amount of static typing. For example, the `cdr` of a list is guaranteed at compile time to be a list, but the child of a tree is not guaranteed to be a tree (the types referred to above are assumed to include `null`, as in Java). The natural question is whether we can use Java generics or guarantee our entire program type safe at compile time. The answer to this depends on the specifics of

Java generics; instead, one can ask whether type safety can be guaranteed assuming sufficiently powerful support for static type checking. One cannot hope to convert *all* type-safe Lisp programs into statically type safe object-oriented programs, because Lisp itself is partly dynamically type checked. Thus, there might be a function that generates a random bit and returns one or the other type accordingly; arbitrarily complex run-time constraints in the program might nevertheless guarantee that it is dynamically type safe.

However, one suspects that reasonable programs like those in the Hughes paper are unlikely to resort to such shenanigans. Indeed, by inspecting the resulting Java code, it is easy to verify that all our functions return the same output type for a given set of input types, provided that all the functions they call have the same property. This should be enough to guarantee static type safety in a sufficiently powerful type system. In practice, the lack of static typing does not seem to be much of an issue, at least at the levels of program complexity herein. Every cast failure that did occur occurred in response to the very first test case.

Lazy evaluation, the Easy way. There is a Right way and an Easy way to do lazy evaluation. The Right way is to evaluate all functions lazily. There are several disadvantages to this approach:

- The syntax becomes cumbersome
- Many benefits of the object-oriented style are lost
- Static type checking becomes even weaker

The Easy way is to evaluate only list accesses lazily. The reason this is meaningful is that list accesses are pretty much the only area where lazy evaluation actually appears to perform less computation than eager evaluation. Of course, this is not true of all programs:

```
(f a b) = (if a b 0)

(print (f (g x) (h x)))
```

The above snippet, when evaluated lazily, avoids computing `(h x)` when `(g x) = 0`. However, this type of lazy evaluation appears rarely in the Hughes examples; almost all the instances that benefit from lazy evaluation are those where list accesses are evaluated lazily. We also note that if it is really vital, a function of two arguments can be expressed as a lazy list of two elements (although this is not completely satisfactory because the caller needs to have some knowledge of how the arguments are used). Thus, our code is drastically simplified by leaving regular method calls as normal eager calls in Java, and only implementing “lazy lists” (as shown in Figure 2.1).

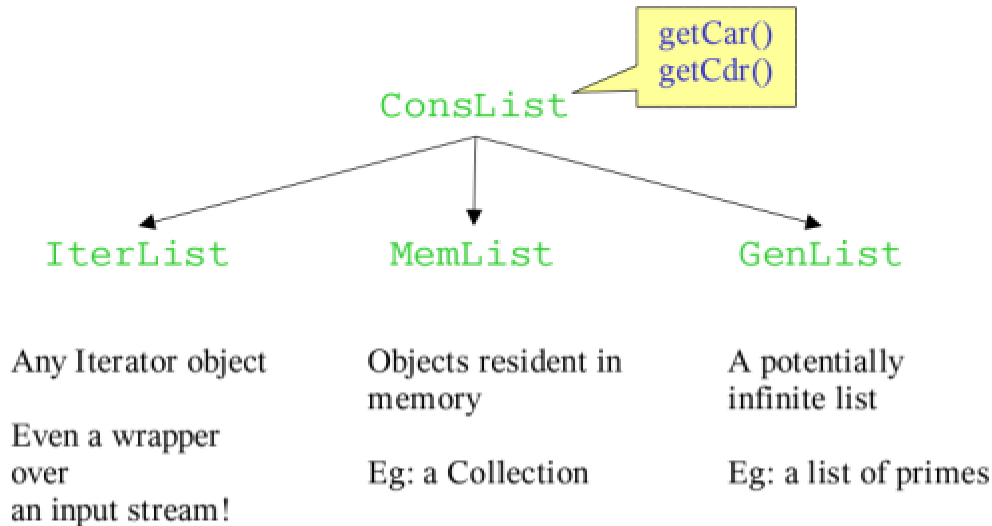

Figure 2.1. *Lazy lists in Java.*

There are a couple of gotchas to our approach, such as not being able to implement `map` in terms of `reduce`, but these are minor and affect only the "wrapper" code.

How does one implement lazy lists in Java? One might naively expect that the Iterator interface is sufficient, because the object returned by `next()` need not necessarily exist in memory until the method is called. However, Java Iterators are *destructive*, in the sense that after two successive calls to `next` (say by executing `iter.next(); if iter.hasNext() { iter.next();}`), the object returned by the first call might very well have become garbage. Another way to think of the destructiveness is that the internal pointer variable is overwritten on each call to `next()`. Because of this, it is no fun to pass Iterator objects around: their behavior depends on how much of the data has been "consumed" by the caller. Lisp lazy evaluation, on the other hand, does not have side effects: two successive calls to `f(input)` are guaranteed to return the same value, no matter how much of the input `f` operates on.

The problem with Java Iterators is inherent to the implementation, and not simply something that could be fixed by changing the interface alone. An Iterator over an input stream, as opposed to one over a memory collection, irreversibly throws away the input elements after calls to `next()`.

Iterators: sources, filters, and sinks. To build non-destructive Iterators in Java, one must first understand the type of Iterators that can exist in a program. Iterators can be thought of has having sources, filters, and sinks, by analogy with Unix pipes. Filters, like `map`, take an iterator as input and are themselves iterators. Sinks, like `print` or `reduce`, "consume" iterators, as they are not themselves iterators. Sources are iterators that are built from non-iterators such as a file input (which can be considered an iterator external to the program), a collection of objects in memory, or no input data at all. Programming languages

usually have built-ins or library functions to make iterators out of input streams or memory objects (such as `java.io.Reader` and subclasses, and the Foreach loop (`for (type object: list)` in Java 1.5). The third type of iterator is called a generator; for example, one that returns the successive prime numbers. Language support for generators could take the form of allowing a function to "yield" instead of return values, implicitly freezing the entire stack frame between invocations. Lack of such support forces the programmer to explicitly save and restore such state as is necessary to return the next element. Of course, generator support is also useful for iterators built *by the programmer* from memory collections, since it avoids the necessity to use an internal pointer variable.

In Lisp, iterators from memory objects and iterators that are generators are both lists. Reading input appears to be destructive, however. We do one better: we unify all three types of iterator sources into one, thus out-Lisping Lisp. The key is the set of classes `ConsList`, `IterList`, `MemList`, and `GenList`. `ConsList` is a general class that behaves like a list in Lisp, except that `car` and `cdr` can only be accessed via method invocations. `ConsList` and all its subclasses behave nondestructively, because the objects returned by `car` and `cdr` are cached. `IterList` subclasses `ConsList` by taking as input an (external) iterator, `MemList` takes a collection of memory objects, and `GenList` is an abstract subclass of `ConsList` that implements a generator.

The crux of our argument is that given this infrastructure, higher order functions and lazy evaluations can be emulated in a manner that is almost syntactic. As stated earlier, this is not something we can objectively prove; we present our Java code as evidence for this assertion.

Other target languages. One lesson learned from this implementation is that Java is perhaps not the ideal vehicle for demonstrating the qualities of object-oriented programming. For example, consider this functional pseudo code:

```
prune 0 (node a x) = node a nil
prune n (node a x) = node a (map (prune (n-1) x)
```

The corresponding Java code is:

```
class PruneOp extends UnaryOp
{
   int depth;

   PruneOp(int theDepth)
   {
      depth = theDepth;
   }

   Object op(Object tree)
   {
      return new PruneTree(depth, (Tree)tree);
   }
```

```
}

class PruneTree extends Tree
{
   int depth;
   Tree tree;

   PruneTree(int theDepth, Tree theTree)
   {
      depth = theDepth;
       tree = theTree;
   }

   Object getRoot()
   {
      return tree.getRoot();
   }

   Object getSubTrees()
   {
      if (depth == 0) return ConsList.empty;
      else return new PruneOp(depth-1).map(tree.getSubTrees());
   }
}
```

Here we can see that only the last two lines do the only actual "computation"; the rest is wrapper code that does nothing more than store, load, and access data. It is in fact almost automatic to derive the Java code from the functional code, but we are nevertheless left with the feeling that the results are sub-par. This does not demonstrate any weakness of object-oriented programming as such, but rather some shortcomings of Java syntax:

- *All functions need to be in classes, and there can be no anonymous objects* — These two are minor inconveniences that increase the verbosity of the code.
- *Tuples are not a built-in type* — In Lisp, currying enables functions of multiple arguments to be treated as functions of a single argument whenever required. Roughly the same effect could be achieved in Java if tuples were a built-in data type. As it is, our choices are to simulate tuples manually using arrays (which leads to very ugly code) or to instantiate the Object representing the function using all but the last parameter, which results in cleaner code but sometimes leads to unnecessary object instantiations. We have chosen the latter alternative.
- *Primitive types are not objects* — Only a minor inconvenience because of "autoboxing".
- *No support for generators* — Such as "yield" in C# or Python.

3. Translation

Rather than showing the functional equivalence between languages by modeling one in the other, translation offers a simpler approach. Although less rigorous than emulation, we can still form a good understanding of how compatible the languages are by looking at the general ways in which constructs in one language can be transformed into another.

Technical Note: The goal of the implementations associated with this section was to provide a natural, object-oriented counterpart to each of the original examples given in the paper, while keeping as close to the style of the original functional versions as possible. Although the implementations are actually written in C#, language-specific features were specifically avoided (for the most part), and all language features that *were* used can be mechanically translated into their Java equivalents. Specifically: 1) In regard to superficial syntactic differences, a) `class C : D` is equivalent to `class C extends D`, b) `class C : I` is equivalent to `class C implements I`, and c) `using library.name;` is equivalent to `import java.library.equivalent;`. 2) In regard to the delegate language shortcuts (used to pass the functions operated on by differentiation and integration, which are secondary to the actual algorithms), delegate declarations (delegate <method declaration>;) may be replaced by the equivalent interface I<operation name> { <method declaration>; }, and delegate functions may be replaced by classes implementing the appropriate interface.

Higher order functions. According to the author, one of the main benefits of using a functional language is the ability to create higher order functions: functions that accept other functions as arguments. A straightforward way to implement this in an object-oriented language is to use interfaces that declare a single method of the desired type.

Although the syntax is a bit cumbersome, an interface is necessary because we need a way to declare the type of the functions we wish to accept in the nominal type systems used by object-oriented languages. Moreover, the overhead of declaring exactly what you want in an interface has advantages of its own. Explicitly declaring the functions you need makes it easier for outside parties to understand how to use the program; it provides stricter type support than in poorly typed or inference-typed languages; and it allows for a clearer definition of the intended functionality than an informal naming convention (i.e. using an `IPredicate` object versus the informal convention of appending `p` or `?` suffixes to functions in Lisp).

For example, in the implementations the interface `interface IDoubleGenerator { double Next(); }` was used to represent infinite lists of data (as discussed in the next section). This declares that the object implementing it is infinite and statically types the return value of the `Next` function is `double`. In a more complicated program where different types of generators are used, a better

approach would be to use generics and an interface like `interface IGenerator<T> { T Next(); }`. Of course if generics are not available, then either a proliferation of `I`*`SomeType`*`Generator`s can be used to maintain static typing, or a single generator type with a common base class (or `Object`, in the worst case) along with dynamic casts can be used.

However, for the most part, this simulated function passing though interfaces is not actually necessary since object-oriented languages offer another alternative, that of *inheritance*. By allowing subclasses to override specific parts of a base implementation (and potentially reuse pieces from that implementation), you can create multiple versions of an algorithm in a well-defined way. Using inheritance also limits the scope of a function, which allows a class to keep a method near the algorithm to which it applies. For example, a more general-purpose numeric class might look like:

```
class NumericAlgorithm {
   virtual double NextApproximation(...);
   private bool GoodEnough(...) {
      /*a test like "within" from the paper*/
   }
   public double ComputeValue() {
      double val = /*initial value*/;
      while( !GoodEnough(...) ) {
         val = NextApproximation(...);
      }
      return val;
   }
}
```

Since the `NextApproximation` method would not be useful to other classes, there is no reason to extend its scope outside of `NumericAlgorithm` and its subclasses. (Of course, if a function *were* generally applicable, then the interface method would be a better approach.)

Lazy evaluation. The other main benefit of functional languages is lazy evaluation. As with the emulation approach, the simplest way to get lazy evaluation is to implement it only where it is used (in this case, when generating potentially infinite sequences in the paper examples without directly implementing any intermediate state). And again, the object-oriented counterpart to the functional version is quite straightforward: explicitly maintain the intermediate state that lazily evaluated functional languages provide automatically.

Of course, there are several ways to accomplish this goal, but all of these options suffer from one common problem. Because they must manage the intermediate state themselves, rather than relying on the language to take care of the details, there is a greater chance that it will be implemented incorrectly. Still, for the toy problems given in the text, state management is not much of a problem (and it is

not clear how much more complicated a lazily evaluated function could get before it too becomes unmanageable in the functional version).

Besides, lazy evaluation contains hidden dangers of its own. For example, one way to implement an infinite list would be to use the common object-oriented idiom of an enumerable collection (for example, by inheriting from `IEnumerable` or `IIteratable`). This would allow callers to iterate through the values in the collection much like a functional program recursively iterates through the nested `cons` elements in a list. And, just like the functional version, if an "infinite" list is ever accidentally passed to a function like `max` or `sort` that must examine all of the elements contained within it, the program will suddenly cease to function because the "infinite" list is indistinguishable from its finite brethren. In the object-oriented version, it is possible to distinguish between such lists by creating a new `IInfinitelyIteratable` interface that inherits from the original, but this does not solve the problem for built-in library functions that deal only with the built-in interfaces. Although reusing the same types for both finite and non-finite lists allows us to reuse our beloved list processing functions, it does not restrict use to only safe functions in the infinite case.

Instead of reusing the vestigial list structure, one might use a more natural representation for the data, such as a generator. This has the advantage of preventing us from using inappropriate list functions on our data, though it does so by preventing use of *all* such functions. However, a simplistic generator like the `IDoubleGenerator` mentioned previously has another problem: it does not entirely capture the way the data is used in the original functional examples, as calls to `Next` in this version are destructive (i.e. if you pass a reference to your generator to another object, the return value of `Next` will depend on whether the other object called `Next`).

Although the examples from the paper work just fine with the destructive version, it is certainly easy enough to correct. For example, one could use an indexing scheme to specify the desired value, or use a different interface like `interface IDoubleGenerator { double Value; IDoubleGenerator NextGenerator(); }` (creating a safe version of the `car` and `cdr` of lists). However, both of these alternatives have the disadvantage of solving the problem by moving the issue of state management onto the caller (though in an admittedly simplified form).

4. General Tradeoffs

Still, although it is easy to duplicate many of the benefits of functional programming in an object-oriented language, we have yet not touched upon the author's underlying motivation: that the features of functional languages promote the creation of well-structured software. Against the archaic assembly code in the paper's one example of the alternative programming style, functional programming's merits are ludicrously obvious. However, the kinds of modularity offered by object-oriented languages make for a better challenge to functional programming's smug superiority.

Structure and modularity. Most all functional languages define modularity as the ability to encapsulate some concept or unit of work in a function. Now it is true that such fine-grained modularity has advantages of its own, as it facilitates low-level composition of arbitrary bits of functionality, but it brings with it several problems of its own. For one, all but a few noble languages (which allow helper functions to be defined within their bodies and hidden from the outside world) make every function defined a global function. This allows both for greater opportunities for function reuse, and for different parts of the program to step all over each other (or intentionally override their "internal" functions). While storing groups of related functions in files can help to organize a program logically, after a file is loaded its contents are summarily dumped into the executing environment along with every other file in use. (In fact, this is a step down from the modularity support in C, which —ignoring for the moment the language's *many* other deficiencies— gives the programmer control over the external visibility of functions defined in the file.)

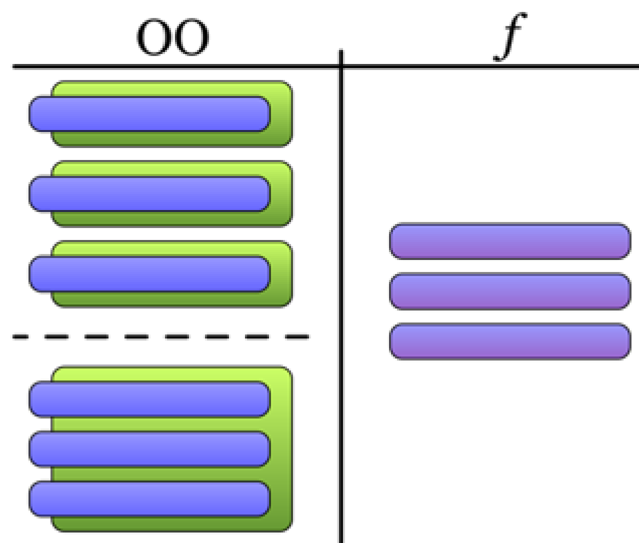

Figure 4.1. *Functions can be defined separately, as in a functional program, or grouped together in a class or interface.*

Compare this to the options for modularity in modern object-oriented languages. At the heart of object-oriented programming is the object (or class) which defines the smallest unit of modularity in a language. Classes can be as small or as large as desired (as shown in Figure 4.1*), and their functionality can be reused and recombined by users calling individual members in an ad-hoc fashion (the functional style), or by creating more specific versions of the class through subclassing. At a higher level, related classes can also be grouped into packages or namespaces, which can be nested hierarchically, and

which allow outside callers to specify exactly which part of the library they want to use (as opposed to loading an entire file *en masse*).

Moreover, although it is certainly possible to define *all* of one's functions in a single class or namespace, such a program would miss out on one of the main benefits of using an object-oriented language: that of organization. Classes and namespaces provide not only the functional organization of functional languages, but also the *logical* organization that these languages lack. As mentioned before, this extra level of organization allows users to specify the pieces of a library they want to use, but more than that it *localizes* the functionality associated with those pieces and makes it available in a context-sensitive way. By decreasing the reach of functions, we make it easier for the user to find exactly what they need (Figure 4.2).

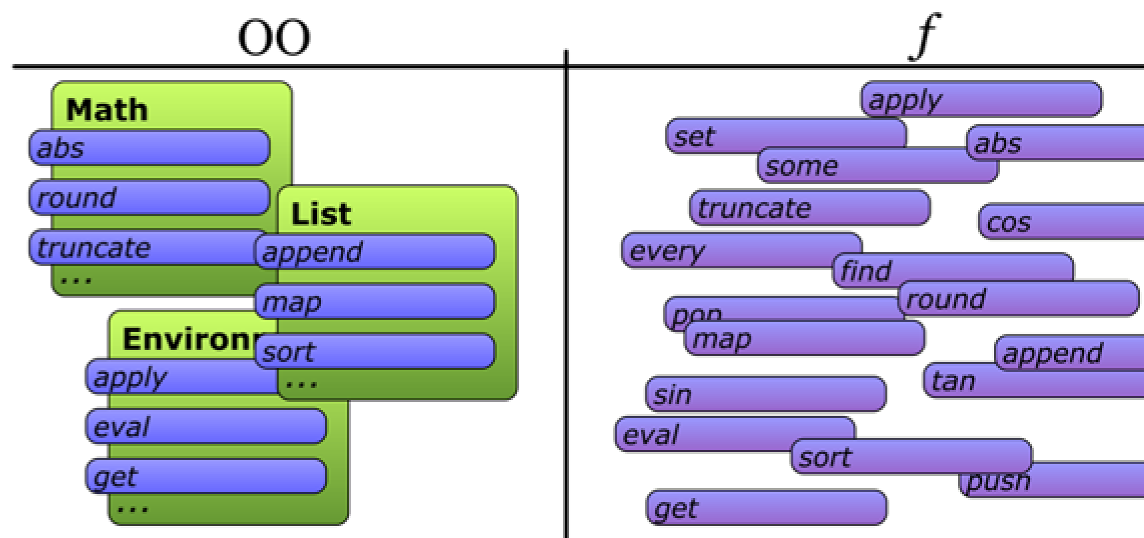

Figure 4.2. *Now what was the name of that function again?*

Of course, such built-in structure has costs of its own. Object-oriented languages not only *allow* functions to be grouped into a class but *force* it. Generally speaking, you cannot define a global function in an object-oriented language (though the benefits of defining such a function globally as opposed to in a globally visible class are questionable); and very few languages allow the direct passing of individual functions beyond passing an instance of the defining object itself. Additionally, although logical organization makes it easy to find related functions when you have the appropriate object in hand, we have actually traded our original problem of function finding for the different (though simpler) problem of finding a desired class among the hierarchy of namespaces.

Abstraction and data management. Another powerful feature that was not touched on directly in the paper was *abstraction*: the ability to present a clean, high-level interface to users by hiding internal details of the implementation. In functional languages, this is normally accomplished by creating a number of functions to create and manipulate black-box chunks of data (which the user

must manage), or by using lexical closure to capture data within a function that can then be passed around as a unit (though, without some complicated interface on the function to allow access to different aspects of the data, this is only useful for simple scenarios).

As we have already seen, object-oriented languages greatly simplify the first scenario by grouping methods related to that data. Besides acting as passive collections of functions, though, objects can also be used for the second scenario by capturing and managing data as well as functions (and in a much more straight-forward manner than the functional alternative, as shown in Figure 4.3).

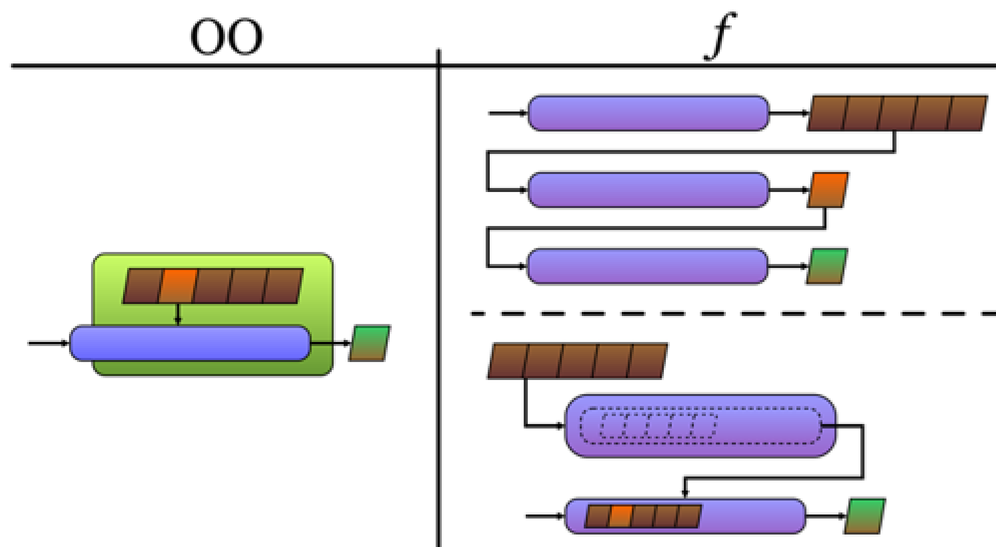

Figure 4.3. *Different methods for managing data.*

Along with interfaces, which allow a function to describe the type of functionality it needs from its inputs without having to specify one specific object type, the ability to create entirely new data structures using objects makes it easier to mold strong abstractions and to make more modular programs. Instead of using an arbitrarily formatted list for *every single data structure* in the program, one can define a new object specific to the needs of the problem at hand. Of course, there is nothing to stop one from using lists *internally* (and thus reusing the list processing libraries), but by encapsulating the data in an object you also gain the freedom (and the language's support) to change that internal representation at any time without affecting outside users. Though, as always, giving authors more control over their code means taking away the flexibility give to outside users to mess with those internals (and if such flexibility is truly desired, the author can always make the data public or emulate the data passing strategies of functional programs).**

5. Conclusion

In this paper we have examined two different approaches to comparing the expressive power of functional and object oriented languages. The first, emulation, attempts to model the constructs of the functional language, providing a wrapper over the object-oriented language on which essentially functional programs can be written. The second, translation, looks at the kind of transformations that can be used to take a functional program and convert it to a natural object-oriented one. We have also examined more general aspects of object-oriented vs. functional programming, in addition to the benefits of functional languages mentioned directly in the paper.

In the end, however, the best approach to programming is probably a mixed one. Just because the features and constructs of functional languages can be easily implemented in an object-oriented fashion does not make the object-oriented approach inherently superior or more powerful; and just because one can program in a solely functional or object-oriented style, to the exclusion of all else, does not make one a better programmer. Both programming styles contain valuable concepts that are more or less appropriate in different situations, and both deserve a place in programmers' repertoire.

Endnotes:

* - Note that, since objects can be considered just collections of functions, in the extreme case where each function is defined in its own class (the upper OO example in Figure 4.1), object-oriented programs are not just *computationally* equivalent to functional programs (in the sense that they are both Turing complete), but are in fact *directly comparable*. I.e. assuming the idiom of immutable data, *any* functional program could be rewritten *directly* as an "object-oriented" one (ignoring for the moment small details like automatic lazy evaluation, which not all functional languages have anyway). It is also interesting to note the comparative ease with which functional languages can be modeled in OO, as opposed to the difficulties encountered when modeling objects functionally, though we shall refrain from further comment.

** - As an aside, data immutability is also a very powerful concept from functional programming, and one that is supported by objects (and, in C# at least, there are several immutable objects defined in the class libraries). Of course, whether it is better to program in a language that frowns on mutable data and then be surprised when someone utilizes it, or to program in a language that has support for both constructs and understands the benefits and consequences of each, is more a matter of opinion than clear-cut advantage for one or the other.