Object-Oriented Programming versus Functional Programming

A Comparison of Concepts

Special Topic in the Lecture on

‘Functional Programming with ML’

by Elmar Ludwig
University of Osnabrück

Winter Term 2001/02

Contents

OOP versus FP

Why Object-Oriented Programming?

Reasons for studying object-oriented programming:

- Learn a different way to think about programming:

  About '*interaction between objects with defined responsibilities*' instead of '*operations (functions) on – possibly shared – data (machine state)*'.
  ($\Rightarrow$ SmallTalk)

- Reduction of complexity by dividing the program into small, reasonably independent and re-usable components, that talk to each other using **only** well-defined interfaces

- Improvement of productivity by using easily adaptable pre-defined software components.

Object-oriented programming is based on many of the fundamental ideas of *structural programming* (modules, information hiding), but also adds new concepts of its own (inheritance, polymorphism).

Fully object-oriented programming languages are *dynamic*:

Some information – like object types or method implementations - does not have to be fully specified (or cannot even be determined) at compile time. Determining this is deferred until run time, which can lead to type errors or invocations of non-existent methods at run time!

Object-Oriented Programming

From *Object-Oriented Programming and the Objective-C Language* [OOP 95]:

"Programming languages have traditionally divided the world into two parts – data and operations on data. Data is static and immutable, except as the operations may change it. The procedures and functions that operate on data have no lasting state of their own; they're useful only in their ability to affect data.

This division is, of course, grounded in the way computers work, so it's not one that you can easily ignore or push aside. Like the equally pervasive distinctions between matter and energy and between nouns and verbs, it forms the background against which we work. At some point, all programmers – even object-oriented programmers – must lay out the data structures that their programs will use and define the functions that will act on the data.

With a procedural programming language like C, that's about all there is to it. The language may offer various kinds of support for organizing data and functions, but it won't divide the world any differently. Functions and data structures are the basic elements of design.

Object-oriented programming doesn't so much dispute this view of the world as restructure it at a higher level. It groups operations and data into modular units called objects and lets you combine objects into structured networks to form a complete program. In an object-oriented programming language, objects and object interactions are the basic elements of design.

Every object has both state (data) and behaviour (operations on data). In that, they're not much different from ordinary physical objects. It's easy to see how a mechanical device, such as a pocket watch or a piano, embodies both state and behaviour. But almost anything that's designed to do a job does too. Even simple things with no moving parts such as an ordinary bottle combine state (how full the bottle is, whether or not it's open, how warm its contents are) with behaviour (the ability to dispense its contents at various flow rates, to be opened or closed, to withstand high or low temperatures).

It's this resemblance to real things that gives objects much of their power and appeal. They can not only model components of real systems, but equally as well fulfil assigned roles as components in software systems."

Comparison to Functional Programming

Basic elements of functional programming are:

- Variables in the mathematical sense (named values)
- Functions in the mathematical sense (mapping values to values)

  The result of a function depends only on its arguments and the context at function definition time.

Basic elements of object-oriented programming are:

- Objects (data) with – typically – non-constant state

  Variables are storage locations in memory that may hold different values at different times.
- Methods (operations) on objects (data) that may or may not change the objects state.

  The result of a method *should* depend only on its arguments and the object's state at method invocation time.

So: Object-oriented programming is basically imperative programming with an additional level of abstraction regarding data and functions.

But:

None the less, you can do functional programming in an object-oriented programming language (even in some imperative programming languages), because:

- Objects (data) *can* have constant state (`final` in Java)

  This leads to the idea of *constant objects* (value objects). Classes like `String` and `Integer` in Java are examples of this.
- Methods (operations) *can* depend only on its' arguments and the context available at method definition time.

How elegant this works out in practice depends heavily on the features supported by a concrete programming language.

Functional Programming in a non FP Language

A small example: `reverse` in Perl

How might a pre-defined '`reverse`' operation on lists be implemented?

It may look like this:

```
sub reverse {
    my @args = @_;

    if (defined $args[0]) {
        (reverse(@args[1..$#args]), $args[0]);
    } else {
        ()
    }
}
```

Or it may look like this:

```
sub reverse {
    my @args = @_;
    my @result = ();

    foreach $arg (@args) {
        @result = ($arg, @result)
    }

    return @result;
}
```

Does it matter?

Functional Programming in a non FP Language

Since a function is really a ‘black box’ (*information hiding*), this actually has two different aspects:

- Implementing a function in functional programming style (implementor’s view)

  A function implemented this way and using only constant data and other ‘functional’ functions looks and works just like its equivalent in a real functional programming language.

- Implementing a function that *behaves* like a function in a functional programming language (user’s view)

  Note that to do this it does **not** have to be implemented in functional programming style!

  Quite a lot of functions (and methods) in typical imperative or object-oriented programming languages are implemented like this (like `strlen()`, `sqrt()` in C, or all methods on `String`-objects in Java).

Functional programming in Java

This was (part of) the definition of the datatype `list` in ML:

```
datatype 'a list = nil
                 | :: of 'a * 'a list;
```

Java needs an explicit constructor (*factory method*) for this:

```
public class List {
    public static List nil = new List();

    public Object head;
    public List tail;

    public static List cons (Object head, List tail) {
        List list = new List();

        list.head = head;
        list.tail = tail;
        return list;
    }
```

Of course it is useful if the value can be printed easily:

```
    public String toString ()
    {
        if (this == nil)
            return "nil";
        else
            return "cons(" + head + ", " + tail + ")";
    }
```

Pattern matching (like in ML) is just 'syntactic sugar' and has basically nothing to to with functional programming, although it does make life a lot easier when dealing with complex data structures.

This can be represented by one or more corresponding predicates and a chain of '`if`'-statements inside the methods:

```
        public boolean iscons ()
        {
            return head != null;
        }
```

And a set of a few typical functions on a list:

```
fun nlength [] = 0
  | nlength (x::xs) = 1 + nlength xs;
```

```
public static int nlength (List l)
{
    if (l == nil)   return 0;
    if (l.iscons()) return 1 + nlength(l.tail);

    throw new IllegalArgumentException();
}
```

```
infix 5 @;   (* append *)
fun ([] @ ys) = ys
  | ((x::xs) @ ys) = x :: (xs @ ys);
```

```
public static List append (List l, List list)
{
    if (l == nil)   return list;
    if (l.iscons()) return cons(l.head, append(l.tail, list));

    throw new IllegalArgumentException();
}
```

```
fun nrev [] = []
  | nrev (x::xs) = nrev(xs) @ [x];
```

```
public static List nrev (List l)
{
    if (l == nil)   return nil;
    if (l.iscons()) return append(nrev(l.tail), cons(l.head, nil));

    throw new IllegalArgumentException();
}
```

```
public static List nrev_evil_evil_evil (List l)
{
    List result = nil;

    for (; l != nil; l = l.tail) result = cons(l.head, result);
    return result;
}
```

This implementation actually is *not* object-oriented at all, which makes it more similar to the ML-version.

Test program (`main`) and output:

```
    public static void main (String args[])
    {
        List l = cons("Hello", cons("World", nil));
        List l2 = append(l, l);
        List lrev = nrev(l2);
        List lrevrev = nrev_evil_evil_evil(lrev);

        System.out.println(l       + " : " + nlength(l));
        System.out.println(l2      + " : " + nlength(l2));
        System.out.println(lrev    + " : " + nlength(lrev));
        System.out.println(lrevrev + " : " + nlength(lrevrev));
    }
}

$ java List
cons(Hello, cons(World, nil)) : 2
cons(Hello, cons(World, cons(Hello, cons(World, nil)))) : 4
cons(World, cons(Hello, cons(World, cons(Hello, nil)))) : 4
cons(Hello, cons(World, cons(Hello, cons(World, nil)))) : 4
```

Functional programming in Java

Somewhat more interesting is the implementation of the *infinite lazy list* (sequence) in Java. Again, first the ML code followed by the corresponding Java code:

```
datatype 'a seq = Nil
                | Cons of 'a * (unit -> 'a seq);

exception Empty;
```

Here, Java needs an `interface` to hold the function signature:

```
public class Seq {
    public static Seq Nil = new Seq();

    public Object head;
    public Succ tail;

    public static interface Succ { Seq succ (); }
    public static class Empty extends RuntimeException {}

    public static Seq Cons (Object head, Succ tail)
    {
        Seq seq = new Seq();

        seq.head = head;
        seq.tail = tail;
        return seq;
    }
```

Again, `toString()` and `isCons()`:

```
    public String toString ()
    {
        if (this == Nil)
            return "Nil";
        else
            return "Cons(" + head + ", fn)";
    }

    public boolean isCons ()
    {
        return head != null;
    }
```

```
fun hd (Cons(x, xf)) = x
  | hd Nil = raise Empty;

fun tl (Cons(x, xf)) = xf()
  | tl Nil = raise Empty;

public static Object hd (Seq s)
{
    if (s.isCons()) return s.head;

    throw new Empty();
}

public static Seq tl (Seq s)
{
    if (s.isCons()) return s.tail.succ();

    throw new Empty();
}
```

Delayed evaluation and a *closure*, realized as an anonymous inner class in Java (that can only have constant context, which is all we want for FP):

```
fun from k = Cons(k, fn() => from (k + 1));

public static Seq from (final int k)
{
    return Cons(new Integer(k),
                new Succ() { public Seq succ () { return from(k + 1); }});
}

fun take (xq, 0) = []
  | take (Nil, n) = raise Subscript
  | take (Cons(x, xf), n) = x :: take (xf(), n - 1);

public static List take (Seq s, int n)
{
    if (n == 0)     return List.nil;
    if (s == Nil)   throw new IndexOutOfBoundsException();
    if (s.isCons()) return List.cons(hd(s), take(tl(s), n - 1));

    throw new IllegalArgumentException();
}

fun squares Nil : int seq = Nil
  | squares (Cons(x, xf)) = Cons(x * x, fn() => squares (xf()));

public static Seq squares (final Seq s)
{
```

```
        if (s == Nil)   return Nil;
        if (s.isCons()) return Cons(new Integer(((Integer) hd(s)).intValue() *
                                                ((Integer) hd(s)).intValue()),
                                    new Succ() { public Seq succ ()
                                        { return squares(tl(s)); }});

        throw new IllegalArgumentException();
    }

    fun Nil @ yq = yq
      | (Cons(x, xf) @ yq) = Cons(x, fn() => (xf()) @ yq);

    public static Seq append (final Seq s, final Seq seq)
    {
        if (s == Nil)   return seq;
        if (s.isCons()) return Cons(hd(s), new Succ() { public Seq succ ()
                                            { return append(tl(s), seq); }});

        throw new IllegalArgumentException();
    }
```

Test program (`main`) and output:

```
    public static void main (String args[])
    {
        Seq s1 = from(1);
        Seq s2 = squares(s1);

        System.out.println(s1);
        System.out.println(tl(s1));
        System.out.println(tl(tl(s1)));
        System.out.println(tl(tl(tl(s1))));
        System.out.println(take(s2, 10));
    }
}

$ java Seq
Cons(1, fn)
Cons(2, fn)
Cons(3, fn)
Cons(4, fn)
cons(1, cons(4, cons(9, cons(16, cons(25, cons(36, cons(49, cons(64,
cons(81, cons(100, nil)))))))))
```

The `Seq` example demonstrates how a function can be returned as a result value. *Functionals* (like `map`), that expect functions as arguments, can be implemented likewise using interfaces and anonymous inner classes.

This is left as an excercise for the reader...

Closures in Object-Oriented Languages

Closures are not only a feature of functional programming languages.

Some object-oriented languages also have *closures* as a language construct and (consequently) allow modifiable context shared between different closures!

Closures in SmallTalk are even used to build control structures [Budd 97]:

```
canAttack: testRow column: testColumn | columnDifference |
    columnDifference <- testColumn - column.
    (((row = testRow) or:
      [row + columnDifference = testRow]) or:
      [row - columnDifference = testRow])
        ifTrue: [ ^ true ].
    ^ neighbour canAttack: testRow column: testColumn.
```

This avoids the special treatment of conditional expressions, which are – as we have seen – not functions in ML (they are in SmallTalk). Of course, this could also be implemented in ML with a little bit of extra work for the programmer.

Lisp handles this in the same way as SmallTalk(?)

Datatypes and Type Polymorphism

Building (abstract) complex data types by combining elements of already defined types is a common feature of both functional and imperative (object-oriented) programming languages.

Much more interesting is the concept of type polymorphism:

A function (or method) that does not need to know the specific type of a parameter value can leave it unspecified (or not fully specified). Then, only those operations may be performed on the value that do not themselves depend on knowlegde of the concrete type.

This is implemented using type variables in ML, so you can write:

```
fun id x = x;
```

Type polymorphism is also one of the central features of object-oriented programming and can be much more flexible than the ML system, but cannot guarantee that errors are detected at compile time!

The function above written in SmallTalk would look like this:

```
id: x | ^ x.
```

Types in OO languages can define *behaviour* (a set of implemented operations), and parameters can be restricted to objects that conform to this behaviour, which is checked at compile time or run time. (⇒ `instanceof` in Java, `respondsTo:` in SmallTalk and `conformsTo:` in Objective-C). ML only has the – not very general – concept of equality types here.

There are more strictly typed object-oriented programming languages (like Java, comparable to ML, but no type inference), full untyped object-oriented programming languages (‘everything is an object’) like SmallTalk, and mixed forms that allow the user to specify a type if appropriate (one example for this is Objective-C, an object-oriented dialect of ANSI-C with OO-extensions taken from SmallTalk).

Functors versus Interfaces

Functors in ML are used to make module implementations independent of each other (which is fundamental to building replaceable structures).

This can be used similar to *classes* in Java (signatures correspond to *interfaces* in this model). Functors however, still suffer from the same limitations regarding type polymorphism:

A functor has to be instantiated for each concrete structure type, and in this process becomes *specific to this type*. You can create a (finite) set of functors for different implementations of a signature, but you cannot instantiate a functor that works with *arbitrary* implementations of this signature (this is similar to ML lists, which can hold only one type of values for any given list).

Functor example in ML (taken from the class notes):

```
signature A =
  sig
    type 'a t;
    val bottom: 'a t;
    val next: 'a t -> 'a t;
  end;

structure As =
  struct
    type 'a t = int list;
    val bottom = [];
    fun next (x) = 1::x;
  end;

functor TestA (At : A) =
  struct
    fun tst a = At.next a;
  end;

structure TestAs = TestA (As);
> [...]
> structure TestAs : {val 'a tst : int list -> int list}
```

Functors versus Interfaces

The same example implemented in Java (note that dummy instances need to be created because interfaces cannot declare static methods, i.e. functions, in Java):

```
public class Functor {
    public static interface A {
        List bottom ();
        List next (List l);
    }

    public static class As {
        public List bottom ()
        {
            return List.nil;
        }

        public List next (List l)
        {
            return List.cons(new Integer(1), l);
        }
    }

    public static class TestA {                  // uses A
        public static List tst (A sig, List l)
        {
            return sig.next(l);
        }
    }

    public static void main (String args[])
    {
        class Asi extends As implements A {}
        A as = new Asi();

        System.out.println(TestA.tst(as, TestA.tst(as, as.bottom())));
    }
}
```

```
$ java Functor
cons(1, cons(1, nil))
```

Bibliography

[OOP 95] Don Larkin, Greg Wilson: Object-Oriented Programming and the Objective-C Language, NeXT Software Inc. 1995

[Budd 97] Timothy Budd: An Introduction to Object-Oriented Programming, Addison Wesley 1997