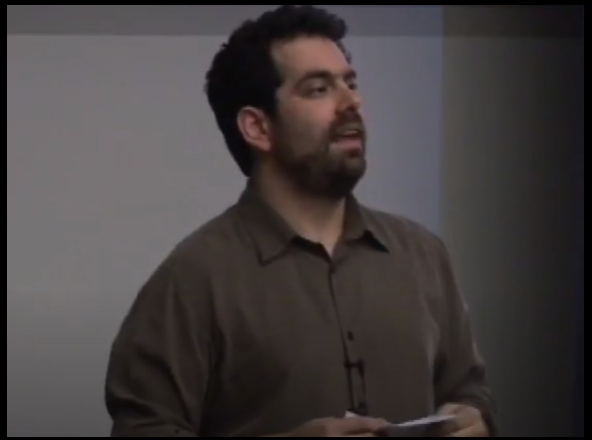
Preamble: Problems with Inheritance



When we first tried switching over from VB to C#, one of the most disturbing features of the language for the partners who read the code was inheritance. They found it difficult to figure out which implementation of a given method was being invoked from a given call point, and therefore, difficult to reason about the code…. At Jane Street, we almost never use objects and never use inheritance. We use standard functional programming techniques and code reviewers find that style more comprehensible.

Caml trading – experiences with functional programming on Wall Street

Dynamic vs. Static Dispatch

```
class C1 {

    public int m1() {
        return 1;
    }

    public int m2() {
        return this.m1();
    }
}

class C2 extends C1 {

    public int m1() {
        return 2;
    }

    public int m3() {
        return super.m1();
    }
}
```

```
public class Example {

    public static void main(String [] args) {

        C1 o1 = new C1();
        C2 o2 = new C2();

        System.out.println(o1.m1());
        System.out.println(o2.m1());

        System.out.println(o2.m3()); // ???
        System.out.println(o2.m2()); // ???
    }
}
```

https://github.com/simondlevy/DynamicDispatch

“Syntactic Noise”

```
public  abstract class Expr<T> {

  public interface Evaluator<T> { boolean evaluate(T value); }
  public abstract boolean eval(Evaluator<T> evaluator);

  public class True<T> extends Expr<T> {
    public boolean eval(Evaluator<T> evaluator) { return true; }
  }
  public class False<T> extends Expr<T> {
    public boolean eval(Evaluator<T> evaluator) { return false; }
  }
  public class Base<T> extends Expr<T> {
    public final T value;
    public Base(T value) { this.value = value; }
    public boolean eval(Evaluator<T> evaluator)
    { return evaluator.evaluate(value); }
  }
  public class And<T> extends Expr<T> {
    public final Expr<T> expr1;
    public final Expr<T> expr2;
    public And(Expr<T> expr1, Expr<T> expr2)  {
      this.expr1 = expr1;
      this.expr2 = expr2;
     }
    public boolean eval(Evaluator<T> evaluator) {
      return expr1.eval(evaluator) && expr2.eval(evaluator);
     }
  }
  public class Or<T> extends Expr<T> {
    public final Expr<T> expr1;
    public final Expr<T> expr2;
    public Or(Expr<T> expr1, Expr<T> expr2)  {
      this.expr1 = expr1;
      this.expr2 = expr2;
     }
    public boolean eval(Evaluator<T> evaluator) {
      return expr1.eval(evaluator) || expr2.eval(evaluator);
     }
  }
  public class Not<T> extends Expr<T> {
    public final Expr<T> expr;
    public Not(Expr<T> expr) { this.expr = expr; }
    public boolean eval(Evaluator<T> evaluator)
     { return !expr.eval(evaluator); }
  }
}
```

Expression evaluator in Java

```java
public  abstract class Expr<T> {

  public interface Evaluator<T> { boolean evaluate(T value); }
  public abstract boolean eval(Evaluator<T> evaluator);

  public class True<T> extends Expr<T> {
    public boolean eval(Evaluator<T> evaluator) { return true; }
  }
  public class False<T> extends Expr<T> {
    public boolean eval(Evaluator<T> evaluator) { return false; }
  }
  public class Base<T> extends Expr<T> {
    public final T value;
    public Base(T value) { this.value = value; }
    public boolean eval(Evaluator<T> evaluator)
    { return evaluator.evaluate(value); }
  }
  public class And<T> extends Expr<T> {
    public final Expr<T> expr1;
    public final Expr<T> expr2;
    public And(Expr<T> expr1, Expr<T> expr2)  {
      this.expr1 = expr1;
      this.expr2 = expr2;
     }
    public boolean eval(Evaluator<T> evaluator) {
      return expr1.eval(evaluator) && expr2.eval(evaluator);
     }
  }
  public class Or<T> extends Expr<T> {
    public final Expr<T> expr1;
    public final Expr<T> expr2;
    public Or(Expr<T> expr1, Expr<T> expr2)  {
      this.expr1 = expr1;
      this.expr2 = expr2;
     }
    public boolean eval(Evaluator<T> evaluator) {
      return expr1.eval(evaluator) || expr2.eval(evaluator);
     }
  }
  public class Not<T> extends Expr<T> {
    public final Expr<T> expr;
    public Not(Expr<T> expr) { this.expr = expr; }
    public boolean eval(Evaluator<T> evaluator)
     { return !expr.eval(evaluator); }
  }
}
```

```
type 'a expr = | True
               | False
               | And  of  'a expr * 'a  expr
               | Or   of  'a expr * 'a  expr
               | Not  of  'a expr
               | Base of  'a

let  rec eval eval_base expr  =
   let  eval' x = eval eval_base x in
   match expr with
   | True  -> true
   | False -> false
   | Base base  -> eval_base base
   | And  (x,y) -> eval' x && eval' y
   | Or  (x,y)  -> eval' x || eval' y
   | Not  x     -> not (eval' x)
```

… in OCAML