ML vs. OO

Close connections between
ML-style programming and OO programming

- datatype ≈ class hierarchy
- function with cases and patterns over datatype constructors ≈ methods in same generic function
- finding matching pattern ≈ method lookup

```
datatype Point
  = CartPoint of {x:real, y:real}
  | PolarPoint of {rho:real, theta:real}
fun getX(CartPoint{x=x,y=_}) = x
  | getX(PolarPoint{rho=rho,theta=theta}) =
     rho * Math.cos(theta)
(* getY is similar *)
fun plus(p1, p2) =
  CartPoint{x=getX(p1)+getX(p2),
            y=getY(p1)+getY(p2)}

- plus(PolarPoint{rho=3.0,theta=1.0},
=        CartPoint{x=3.0,y=4.0});
val it = CartPoint {x=4.62,y=6.52} : Point
```


Some differences

Advantages of OO programming:

- class hierarchies can be extended with new subclasses, without modifying any existing code
- overriding methods can be written in new subclasses, without modifying any existing code
- any class can be used as a type, not just the root class

Advantages of ML programming:

- can define new functions that pattern-match on constructors of existing datatypes, without modifying any existing code
- can pattern-match on multiple arguments simultaneously
- can pattern-match on subpieces of arguments

Can we design a language that supports best of both worlds?
Some attempts:

- Pizza: add datatype-like constructs to Java
- MultiJava: add open classes & multiple dispatching to Java
- O'Caml: add classes, methods, inheritance, etc. to ML
- EML: add OO extensibility to ML

The EML approach

Goals:

- preserve ML flavor
- also allow OO-style programming
- don't force a choice between functional & OO styles

Key ideas:

- allow datatypes to be extensible
- allow functions to be extended with new cases
- introduce rules that allow these extensions to be typechecked module-by-module, safely

Open issues:

- type inference in presence of subtyping
- multiple inheritance

EML now being refined and formalized

Example EML code

Previous ML code for Points is legal EML code

By default:

- datatype root is an abstract class
- datatype constructors are concrete subclasses

Can extend previous code with a new "subclass":

```
extend datatype CartPoint
  = CartPoint3D of {z:real}

fun getZ(CartPoint3D{x=_,y=_,z=z}) = z

extend fun plus(CartPoint3D{x=x1,y=y1,z=z1},
                CartPoint3D{x=x2,y=y2,z=z2}) =
  CartPoint3D{x=x1+x2, y=y1+y2, z=z1+z2}
```

Multiple dispatching

EML can pattern-match on the run-time “class”/“constructor” of multiple arguments

Normal OO languages can’t do this: they dispatch only on the run-time class of the receiver

- static overloading on static argument type less flexible than dispatching on run-time argument class

Leads to problems with e.g. “binary methods” like plus, or whenever the algorithm to run depends on the class of more than one argument

A solution: allow dispatching on arguments other than the receiver

- methods with multiple dispatching called multimethods

Languages with multimethods:

- Common Lisp, Dylan
- Cecil
- Parasitic Java, MultiJava

MultiJava

MultiJava: a backward-compatible extension of Java

Adds

- multimethods
- open classes

Compiles to regular JVM bytecodes

Interoperates seamlessly with existing Java

To indicate which arguments desire dynamic dispatching, add `@Class` modifier

Example of multiple dispatching

```
public abstract class Point implements POINT {
  ...
  public POINT add(POINT p) {
    return new CartPoint(x() + p.x(),
                         y() + p.y()); };
};
public abstract class Point3D
    extends Point implements POINT3D {
  ...
  public POINT add(POINT@Point3D p) {
    return new CartPoint3D(
      x() + p.x(), y() + p.y(), z() + p.z()); };
};

POINT p2d = new CartPoint(3,4);
POINT3D p3d = new CartPoint3D(3,4,5);
POINT p3d_p = p3d;

p3d.add(p3d);      // calls add(PT@Pt3D) in Point3D
p3d_p.add(p3d_p);  // calls add(PT@Pt3D) in Point3D
p3d_p.add(p2d);    // calls add(PT) in Point
p2d.add(p3d);      // calls add(PT) in Point
```


Example of open classes

Idea: allow new methods to be added to existing classes from the outside, without modifying any existing code

- can do multiple dispatching on top-level methods, too

```
// in file sub.mj:
package Geometry.Points;
public POINT Point.sub(POINT p) {
  return new CartPoint(x() - p.x(),
                       y() - p.y());
};
public POINT Point3D.sub(POINT@Point3D p) {
  return new CartPoint3D(x() - p.x(),
                         y() - p.y(),
                         z() - p.z());
};

// in file client.mj:
import Geometry.Points.*;
POINT p = new CartPoint(3,4);
POINT3D p3d = new CartPoint3D(3,4,5);
p3d.sub(p3d);    // calls Point3D.sub(PT@Pt3D)
p3d.sub(p);      // calls Point.sub(PT)
p.sub(p3d);      // calls Point.sub(PT)
```