# Lecture:
# EML and Multimethods

15-312: Foundations of Programming Languages
Jason Reed (jcreed+@cs.cmu.edu)

November 4, 2004

## 1 Object-Oriented Programming

There are several aspects of the *object-oriented* programming style that a proponent of it might point to as centrally important. It advocates achieving modularity of programs through bundling all of the data and behavior of something that can be thought of as an *object*; it encourages reuse of code through *inheritance*; it makes claims on abstraction by use of *dynamic dispatch*. The way a message sent to an object is handled need not be known by the outside world: the object itself 'knows' how to handle it.

Programmers used to programming in a functional style in type systems that support ML-style modules may well have different ideas about what constitutes a natural organization of code and data, and how best to reuse code. But even without getting into arguments about what 'natural' means, we can at least point to one (somewhat) less controversial idiom that many OO languages easily support, which is not as easily (or at least not in the same way) codable in a language like ML.

## 2 Extensibility

### 2.1

The idea in question is *data extensibility*. In, for example, Java, suppose you declare an abstract class and some number of concrete classes.

```
abstract class Exp {
 // return result of substituting v
 // for x in this expression
 Exp subst(Exp v, Int x);
}
...
class Apply extends Exp {
  Apply(Exp e1, Exp e2);
  ... // implement subst
}
...
class Fn extends Exp {
  Fn(Exp e);
  ... // implement subst
}
... // more kinds of Exp
```

Here `Exp` is the abstract class that sits at the top of a part of the class hierarchy that describes the syntax of some language of, perhaps a compiler. It declares methods for operations that the client of this code wants to perform — substitution, for example. To describe the various possible ways of making an `Exp`, we create concrete subclasses of `Exp`, and implement its methods.

That the language supports data extensibility is just the fact that if, at a later time, we decide that we want to expand the old datatype with a new construct, say `IsHalting`, then the changes we need to make to the old code consist of simply adding a new class, with new method implementations:

```
class IsHalting extends Exp {
  IsHalting(Exp e);
  ... // implement subst
}
... // more kinds of Exp
```

The important things to notice here are that (a) the modifications are 'all together' in one place, in one file, and (b) they are not changes to existing library code. As a client of library code that defines the datatype `Exp`, we may extend it by adding new kinds of `Exp`s in our own code.

In contrast, ML (with the exception of the `exn` type) requires that variant datatypes give all of their branches at once, and does not permit extension. Were we to implement the example above in ML, we would start with something like

```
structure Syntax :> SYNTAX =
struct
 datatype exp = Apply of exp * exp
              | Fn of exp
              | ...
 val subst (e : exp, v : exp, x : int) = ...
end
```

To add `IsHalting`, we would have to go in and actually change the original datatype declaration:

```
structure Syntax :> SYNTAX =
struct
 datatype exp = Apply of exp * exp
              | Fn of exp
              | IsHalting
              | ...
 val subst (e : exp, v : exp, x : int) = ...
end
```

Moreover we would have to add another case to the function `subst`, and to any other functions that Syntax might define. Even worse, there might be functions that take arguments of type `exp` outside the `Syntax` structure as well! The changes required to our code could be wildly discontiguous, spanning many files. Depending on how much attention the programmer (or her coworkers) pay to eliminating all nonexhaustive match warnings from a programming project, it may be quite difficult to make sure all functions have been extended appropriately.

### 2.2

Java programmers can take data extensibility for granted, while ML programmers find workarounds, or else just tolerate changing datatype declarations when they must, and hunting down function cases to extend. But

there is a sort of extensibility that ML hackers take for granted that Java hackers symmetrically must go to some pains to acheive: *functional extensibility*. If instead of adding new sorts of data to an existing program what we want to do is add new behavior to existing datatypes, then the difficulty of the task depends on what tools the languages gives us.

Suppose for definiteness that we want to take our compiler above and add an interpreter to it. In the ML case, we're perfectly capable of writing a separate module with a function `eval` like so:

```
structure Eval :> EVAL = struct
 val eval (e : exp) = case e of
                        (Apply(e1,e2)) => ...
                      | (...) => ...
end
```

Note again the advantages we have here: (a) the modifications are 'all together' in one place, in one file, and (b) they are not changes to existing library code. If we write a case for every branch of the type `exp`, then the function works on any `exp` that comes its way. The compiler can provide accurate warnings as to whether our case analysis is nonexhaustive, redundant, or correct.

If we want to do the same thing in Java, then we have at least two obvious options, neither necessarily pleasant. One works only if we have access to the original library, or to its authors. We can add ourselves, or beg the authors to add, a new method to the superclass `Exp`, and implement it in all of the subclasses. This means making many changes in many scattered places, and it is vulnerable to certain kinds of errors. We may implement the method for a subclass $C$ of `Exp` and forget to implement it for a subclass $D$ of $C$: if the inherited code is not appropriate for $D$, then we have failed to make enough changes, but there is no way the compiler can tell us. If a library's API changes because at one of its users' behest, other users may be suddenly stuck with broken code because they don't implement the newly added methods to the abstract superclass.

The other apparent option is writing a static function which contains a big `if-elseif-else` full of `instanceOf` tests. Here we are again capable of leaving out cases in ways the compiler can't detect and sentencing ourselves to unexpected runtime errors. Also, in the pursuit of functional extensibility, we've thrown out convenient data extensibility, at least for the purpose of this one new function. For if we create a new class later, we cannot implement its `eval` case as a method (nor will the compiler know that

we should implement `eval` for it all!) but instead we must hunt down the mass of `instanceOf` tests in the static `eval` function and add a new case.

# 3 EML

## 3.1

ML beats Java on functional extensibility, and Java beats ML on data extensibility. Can't we all just get along, somehow? EML is an attempt to compromise and get both kinds of extensibility at once. It is, in a sense, both a functional language and an object-oriented language. We're going to discuss a simplified version here, but if you want to read the original paper, you can find it at

`http://www.cs.ucla.edu/todd/research/icfp02.html`

Before we get to how EML supports extensibility, we have to discuss one other feature, since it is necessary to understand many aspects of EML's syntax and semantics. EML, as an object-oriented language, has a feature that is not present in Java, but which is present in quite a few other OO languages, called *multimethods*. It is a natural generalization of the dynamic dispatch found in Java. Where a method call `e1.subst(e2,x)` in Java dispatches on the run-time tag of only one of its inputs, namely `e1`, a method call in a language with multimethods can dispatch on the tags of a tuple of arguments all at once. This would be useful if we wanted substitution to do something different depending on what kind of expression `e2` was.

Because multimethods allow dispatch over lists of objects instead of just single objects, we can dispense with the idea that a method is called 'on' a single object. Instead the syntax is closer to an ordinary function call in a functional language. The semantics of the call are still drawn, however, from the object-oriented paradigm: which code gets called as a result of the method invocation depends on the run-time tags of the arguments.

## 3.2

The syntax of the programs in the new language (which again we construe as an extension of MinML) consists of *declarations* in addition to expressions and values. We also have a notion of *classes* in addition to *types*. As with record field labels $\ell$, we assume there to be infinite supplies of class names $C$ and method named $m$. We have that every class is a type, but not every

type is a class: we still have all of our old types like `int` and $\tau_1 \times \tau_2$ and so on. Among classes, there is a notion of *subclassing*, which is determined by which classes are actually declared by the programmer to extend one another.

We have new expressions for constructing objects, which are represented as tagged records $\{C : \bar{\ell} = \bar{e}\}$ with tag $C$ and fields $\bar{e}$, a deconstructor for projecting out object fields (as we did with records) and a way to call methods. Formally the language of classes, types, and expressions is extended as follows:

$$
\begin{array}{rlll}
\textit{Classes} & C & & \\
\textit{Types} & \tau & ::= & \cdots \mid C \\
\textit{Values} & v & ::= & \cdots \mid \{C : \bar{\ell} = \bar{v}\} \\
\textit{Expressions} & e & ::= & \cdots \mid \{C : \bar{\ell} = \bar{e}\} \\
 & & & \mid \#\ell(e) \\
 & & & \mid \texttt{call}\, m(\bar{e})
\end{array}
$$

with the usual conventions about $\bar{\ell} = \bar{e}$, etc.

Now the declarations of the language allow us to create classes, create methods, and implement (i.e. 'extend') methods:

$$
\begin{array}{rlll}
\textit{Declarations} & d & ::= & \cdots \mid [\texttt{abstract}]\, \texttt{class}\, C\, [\texttt{extends}\, C'] \\
 & & & \quad \texttt{of}\, \{\bar{\ell} : \bar{\tau}\} \\
 & & & \mid \texttt{method}\, m(\bar{C}) : \tau \\
 & & & \mid \texttt{extend}\, m(\bar{x} : \bar{C}) = e
\end{array}
$$

A declared class may or may not be *abstract* (i.e. disallow instantiation of itself, and only allow subclassing) may or may not extend (i.e. declare itself a subclass of) another class, and has a set of *fields* $\bar{\ell}$ with types $\bar{\tau}$. A method has a tuple of argument classes $\bar{C}$, and a return type $\tau$. When we implement a case of a method — that is, a piece of code that may run when the method is invoked, depending on the run-time tags of the arguments given — we specify names and types for all of its arguments $\bar{x}$, and provide a function body $e$ in which all the variables $\bar{x}$ are bound.

A program in EML consists of a list of *modules* and an expression to be evaluated. A module for our purposes is just a container for a group of declarations. It is something of the following form:

```
module
 decl
 decl
 ...
 decl
end
```

### 3.3

Before we get to the typing and evaluation rules, let's just take a quick look at some EML code that goes precisely where neither ML nor Java dare tread: pulling off data and functional extensibility at the same time.

Remember we started with a datatype that represented expressions, and a function that performed substitution. In EML we would write this as a module

```
module
 abstract class Exp of {}
 class Apply extends Exp of {e1 : Exp, e2 : Exp}
 class Var extends Exp of {n : Int}
 class Fn extends Exp of {body : Exp}
 ...

 method subst(Exp, Exp) : int -> Exp
// call subst(e, v) x ==> {v/x} e

 extend subst(e : Apply, v : Exp) = fn x =>
 {Apply: e1 = call subst(#e1 e, v) x,
         e2 = call subst(#e2 e, v) x}

 extend subst(e : Var, v : Exp) = fn x =>
   if #n e = x then v else e

 extend subst(e : Fn, v : Exp) = fn x =>
  {Fn: body = call subst(#body e) (x+1)}
 ...
end
```

Just like in Java, we have a abstract (uninstantiable) class of expressions, and one concrete class for every variety of expression. We declare a method that takes two `Exp` arguments, and returns a function that takes an int and returns an `Exp`. We might also view this as a partially curried function taking three arguments. We implement the method by giving its behavior on particular cases of its arguments' run-time tags. If the first argument happens to be an `Apply` and the second (as it will inevitably be, as long as the program is well-typed) is an `Exp`, then the first piece of code will execute. Similarly if the first argument is a `Fn` then the second piece of code will execute.

If we want data extensibility, then in another module we can write

```
module
 class IsHalting extends Exp of {body : Exp}

 extend subst(e : IsHalting, v : Exp) = fn x =>
 ...
end
```

If we want functional extensibility, then in another module we can write

```
module
 method step(Exp) : Exp

 extend step(e : Apply) = ...
 extend step(e : Fn) = ...
 extend step(e : Var) = raise Stuck
 ...
end
```

And if we want both extensions at once, we can write a fourth module completely separate from the above three that fills in the evaluation case for `isHalting`:

```
module
 extend step(e : isHalting) = ...
end
```

From the point of view of ML, what EML gives you is the ability to add new datatypes and new function cases at places in your code far away from the original declarations of those datatypes and functions. From the point of view of Java, what EML allows is the addition of new methods to old class hierarchies (again, far away from the original codebase) in a way that fits smoothly with the existing dispatch mechanism.

### 3.4

For the typing rules, we assume $Decls$ to be the set of existing declarations. Every time a module is encountered and determined to be well-formed, its declarations are imperatively added to the set $Decls$. For now, well-formedness of modules will depend on all of the previous parts of the program up to that point. In next lecture we will discuss how to make these checks modular, i.e. local to the current module.

An object expression is well-typed if its tag represents a declared class, and its fields are all well-typed.

$$\frac{\texttt{class}\ C\ \texttt{of}\ \{\bar{\ell} : \bar{\tau}\} \in Decls \qquad \Gamma \vdash e_i : \tau_i \quad \text{(for all } i\text{)}}{\Gamma \vdash \{C : \bar{\ell} = \bar{e}\} : C}$$

If the class $C$ subclasses another class $C'$, then the left-over fields must form a valid object of class $C'$.

$$\frac{\texttt{class}\ C\ \texttt{extends}\ C'\ \texttt{of}\ \{\bar{\ell} : \bar{\tau}\} \in Decls \qquad \begin{array}{c}\Gamma \vdash e_i : \tau_i \quad \text{(for all } i\text{)} \\ \Gamma \vdash \{C' : \bar{\ell}' = \bar{e}'\} : C'\end{array}}{\Gamma \vdash \{C : \bar{\ell} = \bar{e}, \bar{\ell}' = \bar{e}'\} : C}$$

A projection expression is well-typed if its body is an object of a class with the indicated field.

$$\frac{\ell : \tau \in Fields(C) \qquad \Gamma \vdash e : C}{\Gamma \vdash \#\ell(e)}$$

where $Fields$ is defined by

$$\frac{\texttt{class}\ C\ \texttt{of}\ \{\bar{\ell} : \bar{\tau}\} \in Decls}{Fields(C) = (\bar{\ell} : \bar{\tau})}$$

$$\frac{\texttt{class}\ C\ \texttt{extends}\ C'\ \texttt{of}\ \{\bar{\ell} : \bar{\tau}\} \in Decls}{Fields(C) = (\bar{\ell} : \bar{\tau}, Fields(C'))}$$

A method call is well-typed as long as all of its arguments are:

$$\frac{\texttt{method}\ m(\bar{C}) : \tau \in Decls \qquad \Gamma \vdash e_i : C'_i \quad \text{(for all } i\text{)} \quad \bar{C}' \leq \bar{C}}{\texttt{call}\ m(\bar{e}) : \tau}$$

where here $\leq$ is the declared subclass relation defined by

$$\frac{\texttt{class}\ C\ \texttt{extends}\ C'\ \texttt{of}\ \{\bar{\ell} : \bar{\tau}\} \in Decls}{C \leq C'}$$

$$\frac{}{C \leq C}$$

$$\frac{C \leq C' \qquad C' \leq C''}{C \leq C''}$$

Since there are only finitely many classes declared in a given program, introducing a transitivity rule here is not harmful. We can efficiently compute the reflexive, transitive closure of the is-a-direct-subclass relation. The notation $\bar{C} \leq \bar{C}'$ means that $C_i \leq C'_i$ for all $i$.

### 3.5

The operational semantics of EML depend on the idea of a *best match* for a given method call. Suppose we have an object hierarchy consisting of the classes Square and Rect, where Square is a subclass of Rect, and the method `Intersect(Rect, Rect):bool`. If we declare cases for Intersect for all four possibilities of Square and Rect inputs,

```
...
extend Intersect(Rect, Rect) = ...
extend Intersect(Rect, Square) = ...
extend Intersect(Square, Rect) = ...
extend Intersect(Square, Square) = ...
...
```

then we can consider what happens when we invoke the method by writing `call Intersect(e1, e2)` for some expressions `e1, e2`. If `e1` and `e2` are both Squares, then although all four cases match — in the sense that

the arguments given by the method call are subtypes of the arguments required by each case — only the fourth case seems appropriate, since it is the most specific.

We don't want to require exact matches between the run-time arguments and the case's arguments, for if we only wrote

```
...
extend Intersect(Rect, Rect) = ...
...
```

we would still like the code to work on Squares as well by inheritance. So *if* there is a more specific case we will use it, but any maximally specific case that matches is the one we will execute.

Formally, the evaluation rules are comprised of search rules

$$\frac{e \mapsto e'}{\{C : \bar{\ell}^\flat = \bar{v}^\flat, \ell = e, \bar{\ell}^\sharp = \bar{e}^\sharp\} \mapsto \{C : \bar{\ell}^\flat = \bar{v}^\flat, \ell = e', \bar{\ell}^\sharp = \bar{e}^\sharp\}}$$

$$\frac{e \mapsto e'}{\#\ell(e) \mapsto \#\ell(e')} \quad \frac{e \mapsto e'}{\texttt{call}\ m(\bar{v}, e, \bar{e}) \mapsto \texttt{call}\ m(\bar{v}, e', \bar{e})}$$

the reduction rule

$$\frac{}{\#\ell_i(\{C : \bar{\ell} = \bar{v}\} \mapsto v_i}$$

and the dispatch rule

$$\frac{\begin{array}{c}\texttt{extend}\ m(\bar{C}') = e \in Decls \\ \bar{C} \leq \bar{C}' \\ \text{For any other } \texttt{extend}\ m(\bar{C}'') = e' \in Decls \text{ such that} \\ \bar{C} \leq \bar{C}'' \text{ we have } \bar{C}' \leq \bar{C}''\end{array}}{\texttt{call}\ m(v_1\ \texttt{as}\ \{C_1 : \cdots\}, \ldots, v_n\ \texttt{as}\ \{C_n : \cdots\}) \mapsto \{\bar{v}/\bar{x}\}e}$$

This last rule expresses that we dispatch to a method case if (a) the arguments $\bar{C}$ are individually subclasses of the required arguments $\bar{C}'$ of that case and (b) any other case $m(\bar{C}'')$ that also matches the arguments $\bar{C}$ is more general (i.e. less specific) than this case.

**3.6**

The problem with the operational semantics given is that it may get stuck, in one of two ways.

One is that we try invoking a method for which no case is applicable. This happens if we have declared the Intersect method and implemented it only for a pair of Squares, and try to invoke it on a pair of Rectangles (or on a Rectangle and a Square).

The other is that more than one case is applicable, and no single case is the most specific. If what we have implemented is only

```
...
extend Intersect(Rect, Square) = ...
extend Intersect(Square, Rect) = ...
...
```

and we try to invoke Intersect on two Squares, then both cases are applicable, but neither is more specific than the other. It would take an implementation of

```
extend Intersect(Square, Square) = ...
```

to remedy this.

In the next lecture we will discuss EML's method for preventing these kind of run-time errors, in an efficient and local way. Until then, imagine that the type-checker does a global pass over the program, trying by brute force to determine whether these exhaustivity and ambiguity errors can arise.

For exhaustivity, we can simply enumerate all of the possible argument lists that could legitimately be given to a declared method, and check for the existence of a case that handles each one.

For ambiguity, we can, for each method, enumerate every pair of declared cases it has. Suppose the case we have in mind are `extend` $m(\bar{C}) = e$ and `extend` $m(\bar{C}') = e'$. Now for a set of arguments to cause an ambiguity error to occur with these two cases, it would have to match both of them. That is, there would be a list of classes $\bar{C}''$ such that $\bar{C}'' \leq \bar{C}$ and $\bar{C}'' \leq \bar{C}'$. For each position in this list, if there are any common subclasses of $C_i$ and $C_i'$, there is a 'most super' subclass, a subclass highest in the hierarchy. In fact it must be either $C_i$ or $C_i'$. This follows from the fact that we have no

multiple inheritance. Finding this 'most super' subclass is also called finding the *greatest lower bound* of $C_i$ and $C_i'$, which we can write as $C_i \cap C_i'$. It may be, however, if neither $C_i$ nor $C_i'$ is a descendant of the other, that there is no lower bound at all. If this is ever the case, we are safe with respect to this pair of cases, for no list of arguments could ever satisfy both.

Otherwise we can do this for all elements in the list, thus finding $\bar{C} \cap \bar{C}'$. This list of classes, if it exists, is the most general set of arguments that is specific enough to possibly trigger the two cases we began considering. Now we simply check whether $\bar{C} \leq \bar{C}'$ or $\bar{C}' \leq \bar{C}$. If either of these holds, we are safe, for there is no ambiguity. One case is more specific than the other. Otherwise, we throw up a compile-time error, because invoking method $m$ with arguments tagged with $\bar{C} \cap \bar{C}'$ would cause a run-time ambiguity error.