Supplementary Notes on Objects

15-312: Foundations of Programming Languages
Jonathan Aldrich

Lecture 20-21
November 4-6, 2003

In this lecture we examine the design goals and technical properties of object-oriented languages. It is possible to study some features of objects by encoding them in terms of constructs we already know; however, realistic encodings of objects are so complex that it is often simpler to study objects as first-class constructs. Rather than study objects in the context of a completely different language (such as Java) we will look at how objects can be added naturally to already powerful functional languages like ML.

A Brief History of Objects.

The first object-oriented language was Simula 67, developed by Ole-Johan Dahl and Kristen Nygaard (who won the 2002 Turing award for this work). Simula 67 had many of the same concepts as modern object-oriented programming languages, including objects, classes, virtual methods, and inheritance. The motivation for the Simula languages came due to Kristen Nygaard's work on operational research. Nygaard needed a tool for the precise description and simulation of complex systems involving both people and machines. Simula 67 was designed to support both a description of complex systems and a simulation or execution of these systems.

The design goal of modeling real-world systems influenced the features of Simula 67, as well as more recent object-oriented languages. Real-world systems are built of entities that have state and behave in certain ways; these entities are modeled as objects. We often classify real-world entities into hierarchical categories. Each category provides a partial description of the state and behavior of entities in that category: for example, animals

typically move, while plants typically photosynthesize. These classification structures are modeled with classes that describe the state and behavior of object instances. Our classification structures are often hierarchical, so that mammals are a particular kind of animal that is warm-blooded, furry, gives birth to live young, etc. Similarly, object-oriented programming languages allow one class to inherit from another, so that the state and behavior specifications in one class can be reused in its subclasses. Virtual methods are provided so that subclasses can refine the behavior defined in their superclasses.

Most object-oriented languages are imperative. This is not strictly necessary– a number of purely functional object-oriented languages have been defined. However, imperative objects form a natural model of the world, because we usually think of the world as having state that changes from moment to moment. As in the real world, the behavior of an object-oriented system are derived from the interactions between objects, as described by their methods.

The design goals of object-oriented langauges also provide insight into the software engineering tradeoffs in using object-oriented languages. For example, in order to build a useful program, you need to both verify it (build the system right) and validate it (build the right system). Functional languages are based closely on the mathematical theory of computation; thus, they facilitate verification, or building the system right. Object-oriented languages are "messier" in a sense: functions are split into methods, references are all over the place, and object-oriented dispatch makes it hard to tell what code is running. Therefore, it is often harder to verify that an object-oriented system is built right. However, object-oriented languages are very good at modeling the world using abstractions that are close to real-world concepts. Thus, it is often easier to validate object-oriented systems systems–ensuring that the programmers wrote the right system, one that fulfills the needs of users. In practice, of course, we need both verification and validation, and the relative benefits and drawbacks of different languages may depend on the domain in which they are used, as well as many other factors.

EML: Objects in ML.

Unfortunately many object-oriented languages do not take advantage of the benefits of advanced functional programming languages. For example, Java makes it easy to add new classes to a program externally by subclassing from existing classes. However, there is no way to externally add a new

function with cases for each class–instead the function must be split up and added as one method to each relevant class in the system, something that is impossible if the system's source code is not available. In comparison, this task is easy in ML–you simply define a new function with cases for each relevant datatype constructor.

A better language design combines the benefits of both functional and object-oriented languages. Here we follow the design of Extensible ML (EML), a recently-developed language that provides object-oriented features in ML by making datatypes and functions hierarchically extensible. The grammar of EML's class and function declarations is given as follows:

$$
\begin{array}{rcl}
decl & ::= & [\texttt{abstract}]\ \texttt{class}\ C\ [\texttt{extends}\ C'] \\
 & & \texttt{of}\ \{\overline{l} : \overline{\tau}\} \\
 & | & \texttt{fun}\ f : \overline{C} \to \tau \\
 & | & \texttt{extend fun}\ f\ (\overline{x}\ \texttt{as}\ \overline{C}) = e
\end{array}
$$

In this example, we use the overbar notation $\overline{t}$ to denote a tuple of terms that have the form t. We assume that expressions e have the same form as in MinML programs we have already studied. The EML language is declaration-oriented, and there are three forms of declarations: `class` declarations, function signature declarations, and function cases.

A class declaration in EML is an extended form of an ML datatype declaration, where the body of the datatype is an extensible record. Class declarations can be declared `abstract`, meaning that the class cannot be instantiated, only extended. Inheritance is represented by an `extends` clause. Finally, the body of the class is a record mapping labels $\overline{l}$ to types $\overline{\tau}$.

For example, the code below declares three classes. The Point class is abstract; it defines the interface of points, but has no fields and cannot be instantiated. The Cartesian point is a concrete subclass of Point, so it can be instantiated by clients. It declares x and y fields to hold the coordinates. Finally, ColoredPoint extends CartesianPoint, thus inheriting the x and y fields, and adding a new field representing the point's color.

```
abstract class Point of {}
class CartesianPoint extends Point of {x:float, y:float}
class ColoredPoint extends CartesianPoint of {color:Color}
```

Function declarations are split into two parts. A `fun` declaration declares the type of a function from a tuple of classes $\overline{C}$ to a type τ. In the full

EML language, functions can have any argument and result type, but we are simplifying the system for the purposes of modeling it formally.

For example, we could declare the interface of point with the code below. Point declares three operations that get the x and y coordinates and compute the distance between two points. ColoredPoint declares an additional operation to get the point's color; this operation only applies to ColoredPoint and its subclasses, not to Point or CartesianPoint.

```
fun getX : Point -> float
fun getY : Point -> float
fun distance : Point * Point -> float
fun getColor : ColoredPoint -> Color
```

Function cases are declared with the `extend fun` declaration. The `extend fun` keywords are followed with a pattern expression describing the conditions under which the function case is applicable. In our formal treatment, we assume that the pattern is a tuple of bindings of variables to classes; the full EML language permits an arbitrary pattern here. The body of the function e is evaluated in a scope where the variables $\overline{x}$ from the pattern match are bound to the actual parameters of the function.

For example, the following functions define an implementation of the code above for the concrete classes CartesianPoint and ColoredPoint. Note that there is no separate definitions for getX, getY, and distance for ColoredPoint. Instead, the function cases for CartesianPoint and/or Point work equally well for ColoredPoint. Note that we do not have to define function cases for Point, because that class is abstract and can never be instantiated. It may still be convenient to do so, however; for example, in the code below, the implementation of distance can be shared by any subclass of Point, including subclasses that have not yet been defined (PolarPoint, anyone?).

```
extend fun getX (p as CartesianPoint) = #x(p)
extend fun getY (p as CartesianPoint) = #y(p)
extend fun distance (p1 as Point, p2 as Point)
               = sqrt(sqr(getX p1)+sqr(getX p2))
extend fun getColor (p as ColoredPoint) = #color(p)
```

We also need new syntax for creating objects: this syntax the same one we would use to create a ML datatype that is implemented as a record:

$$e \ ::= \ C\{\bar{l} = \bar{e}\}$$

Here we create an object of class C, initializing the fields with the expressions $\bar{e}$. For example, we can create a ColoredPoint object with the following code: `ColoredPoint{x=1,y=3,color=Blue}`.

Object-Oriented and Functional Programming.

In many ways, objects are similar to features provided by other languages. For example, a class is a lot like a constructor in a datatype, and a method in class C is a lot like the case for datatype constructor C in a ML function definition. For example, we could have defined a datatype for Point in ML with the code below. Here the abstract class Point is the name of the datatype, while the concrete subclasses of Point have been converted into constructors for the datatype.

```
datatype Point
    = CartesianPoint of {x:float,y:float}
    | ColoredPoint of {x:float,y:float,color:Color}
```

However, object-oriented languages provide several properties not found in ML or other functional langauges. First, classes and methods are extensible, unlike datatype and function declarations. In object-oriented languages, you can add a new class at any time. One important benefit of this property is that you can extend or build on top of a system by adding new classes even when you cannot modify the system's source code. Each class can include its own implementation for methods inherited from superclasses. For example, the following EML code adds PolarPoint a new kind of Point with its own representation and methods. Note that PolarPoint can be added without changing any existing code.

```
class PolarPoint extends Point of {rho:float, theta:float}
extend fun getX (p of PolarPoint) = rho * cos(theta)
extend fun getY (p of PolarPoint) = rho * sin(theta)
```

In contrast, when you declare a datatype in ML, you have to give all of the constructors in one place, and for each function declared over the

datatype, a case must be given for each constructor at the same location in the source code. Thus, in order to add a new datatype case in ML you have to modify the datatype declaration and all functions over the datatype, which can be a major problem if the source code is not available.

Second, class extension is hierarchical. If class C extends another class B, then C inherits all of the fields and methods from its superclass. Any code written for class B can be customized and reused by class C–and this is true whether or not the author of B intended for the code to be reused. For example, the PolarPoint code reuses the definition of distance written for class Point. Similarly, ColoredPoint reuses all of the code written for class CartesianPoint.

Functional languages like ML also provide support for customized reuse through first-class functions and functors. Compared to these mechanisms, inheritance is more structured than passing around first-class functions, but more lightweight and flexible than writing a functor. Inheritance also typically requires less advanced planning for reuse, since extensibility is "built in" to the language construct.

The hierarchical nature of class extension is also useful in that subclasses can define methods that are not present in their superclasses. In functional languages, this is equivalent to writing a function that is defined on some constructors in a datatype but not others. As we will see, a primary benefit of object-oriented languages is that the type system can ensure that such partial functions are never called on classes on which the function is not defined. For example, the getColor function is only defined on ColoredPoints, and the type system will ensure that it is never invoked on other kinds of Points. There is no way to get this property when using ordinary ML datatypes. However, current CMU research on refinement types is focused in part on providing this benefit in the context of functional languages.

Operational Semantics for EML

EML's operational semantics follows the same rules as in MinML, except for object creation and calls to functions. Object creation is defined in a similar way to record creation. Evaluation proceeds one field at a time; when all the fields are values, then the object expression is a value.

Function calls must be treated differently from MinML because the system must use the actual classes of the arguments to determine which function case to execute. We define function case lookup using an auxiliary function, $lookup$, which chooses the most specific applicable function case

for a given tuple of classes.

$$\frac{e_i \mapsto e_i'}{\begin{array}{c} C\{l_{1..i-1} = v_{1..i-1},\ l_i = e_i,\ l_{i+1..n} = e_{i+1..n}\} \mapsto \\ C\{l_{1..i-1} = v_{1..i-1},\ l_i = e_i',\ l_{i+1..n} = e_{i+1..n}\} \end{array}}\ \textit{Field}$$

$$\frac{\overline{v}\ \ \mathsf{value}}{C\{\overline{l} = \overline{v}\}\ \ \mathsf{value}}\ \textit{ObjVal}$$

$$\frac{\overline{v} = \overline{C\{\ldots\}} \quad \mathit{lookup}(f, \overline{C}) = \overline{x}.e}{f\ (\overline{v}) \mapsto \{\overline{v}/\overline{x}\}e}\ \textit{FnApp}$$

$$\frac{\begin{array}{c} (\texttt{extend fun}\ f(\overline{x}\ \mathsf{as}\ \overline{C'}) = e) \in \mathit{Decls} \qquad \overline{C} <: \overline{C'} \\ (\texttt{extend fun}\ f(\overline{x'}\ \mathsf{as}\ \overline{C''}) = e') \in \mathit{Decls} \Rightarrow (\overline{C'} <: \overline{C''} \wedge \overline{C''} \not<: \overline{C'}) \end{array}}{\mathit{lookup}(f, \overline{C}) = \overline{x}.e}\ \textit{Lookup}$$

Client-Side Typechecking in EML

The typing rules for EML are divided into client-side and implementation-side typechecking. Client-side typechecking is the same kind of typechecking we have been studying up to this point in the class: we check that expressions e have types τ, and that class declarations and function declarations are well formed, written $\mathit{decl}\ OK$. Implementation-side typechecking, discussed in a later section, verifies that each generic function is implemented completely and consistently.

EML's rules for client-side typechecking build on the rules we have been studying for MinML. The rules that are new are given below. Several of the rules refer to Decls, the (global) set of declarations visible in the program.

$$\frac{(\mathsf{fun}\ f : \overline{C} \to \tau) \in Decls}{\Gamma \vdash f : \overline{C} \to \tau}\ \textit{FnTyp}$$

$$\frac{C\ \mathsf{class} \quad C\ \mathsf{concrete} \quad \Gamma \vdash \overline{e} : \overline{\tau} \quad fields(C) = \{\overline{l} : \overline{\tau}\}}{\Gamma \vdash C\{\overline{l} = \overline{e}\} : C}\ \textit{ClsTyp}$$

$$\frac{\overline{C}\ \mathsf{class} \quad \tau\ \mathsf{type}}{\mathsf{fun}\ f : \overline{C} \to \tau\ \ OK}\ \textit{FnOK}$$

$$\frac{(\mathsf{fun}\ f : \overline{C'} \to \tau) \in Decls \quad \overline{C} <: \overline{C'} \quad \overline{x}{:}\overline{C} \vdash e : \tau}{\mathtt{extend}\ \mathtt{fun}\ f\ (\overline{x}\ \mathtt{as}\ \overline{C}) = e\ \ OK}\ \textit{ExtOK}$$

$$\frac{C'\ \mathsf{class} \quad \overline{\tau}\ \mathsf{type} \quad \{\overline{l}\} \cap fields(C') = \emptyset}{[\mathtt{abstract}]\ \mathtt{class}\ C\ [\mathtt{extends}\ C']\ \mathtt{of}\ \{\overline{l} : \overline{\tau}\}\ \ OK}\ \textit{ClsOK}$$

The first rule gives a type to a variable in client code that refers to a function that has been declared. The second rule states that an object of class C is well-typed and has C if C is a class and C is concrete. It also requires that the expressions assigned to the fields of C have types that match the class declaration.

The rule for function declarations ensures that the declared classes and types exist. The rule for function cases checks that the classes in the pattern expression are subtypes of the classes in the function signature. The rule also checks that the body is well-formed, assuming the arguments have appropriate types. The rule for classes just checks that the types and classes referred-to actually exist, and also verifies that the new fields are distinct from the ones being inherited. In a real language, we would allow one field to shadow another, disambiguating them using the class in which the fields were defined.

To be precise, we should also define auxiliary judgments of the form C class and $fields(C)$.

$$\frac{([\texttt{abstract}]\ \texttt{class}\ C\ \ldots) \in Decls}{C\ \mathsf{class}}\ \textit{Class}$$

$$\frac{C\ \mathsf{class}}{C\ \mathsf{type}}\ \textit{ClassType}$$

$$\frac{([\texttt{abstract}]\ \texttt{class}\ C\ \texttt{of}\ \{\bar{l} : \bar{\tau}\}) \in Decls}{fields(C) = \{\bar{l} : \bar{\tau}\}}\ \textit{FieldsBase}$$

$$\frac{([\texttt{abstract}]\ \texttt{class}\ C\ \texttt{extends}\ C'\ \texttt{of}\ \{\bar{l} : \bar{\tau}\}) \in Decls}{fields(C) = \{\bar{l} : \bar{\tau}\} \cup fields(C')}\ \textit{FieldsInherit}$$

The first two rules state that a class exists if it is declared in $Decls$, and that each class defines a type with the same name. The fields of a class are computed based on the union of the inherited and the newly-declared fields. We can define the subtyping relation as the reflexive, transitive closure of the declared inheritance relation:

$$\frac{([\texttt{abstract}]\ \texttt{class}\ C\ \texttt{extends}\ C'\ \ldots) \in Decls}{C <: C'}\ \textit{SubBase}$$

$$\frac{}{C <: C}\ \textit{SubReflex}$$

$$\frac{C <: C' \quad C' <: C''}{C <: C''}\ \textit{SubTrans}$$

Implementation-Side Typechecking in EML

To complete the typing rules for EML, we must give rules for implementation-side typechecking as well. Implementation-side typechecking verifies that each function in the system is completely and consistently implemented. For example, consider the following code:

```
abstract class Shape of  ...
class Rectangle extends Shape of  ...
class Circle extends Shape of  ...
```

```
fun draw : Shape -> unit
extend fun draw (c as Circle) = ...
```

```
fun intersect : Shape*Shape -> boolean
extend fun intersect (c as Circle, s as Shape) = ...
extend fun intersect (s as Shape, r as Rectangle) = ...
```

In the code above, the draw function is incomplete, because there is no case for Rectangle. It's actually fine that there is no case for Shape, because since Shape is an abstract class, and so we can never create a Shape at run time. However, if we create a Rectangle at run time and pass it to draw, we will get a message not understood error. This terminology stems from the object-oriented practice of calling functions "messages" that are sent to objects like Rectangle; in this case, there is no function case for Rectangle in the message, so we say that Rectangle did not understand the message.

Similarly, in the code above, the intersect function is ambiguous, because if the function is invoked with the tuple (c,r), where c is a Circle and r is a Rectangle, then either case of the intersect function applies to the tuple, but neither is more specific than the other. This is known as a message ambiguous error.

The goal of implementation-side typechecking is to statically guarantee that message not understood and message ambiguous errors cannot occur. We will first define a global typechecking algorithm formally, and then discuss informally how to derive a modular typechecking algorithm from the global one. The global algorithm is given by the rule below:

$$\frac{\begin{array}{c}\forall(\texttt{fun}\ f : \overline{C} \to \tau) \in Decls \\ \forall \overline{C'}\ where\ \overline{C'}\ \textsf{concrete}\ and\ \overline{C'} <: \overline{C} \\ lookup(f, \overline{C'}) = \overline{x}.e\end{array}}{Decls\ \ OK}\ \textit{Global-ITC}$$

The rule above checks, for each function f and for each tuple of concrete classes $\overline{C'}$ that subtype the classes declared in the function signature, whether there is a unique function case for that tuple of classes. The same $lookup$ function that is used for dispatching at run time is used to check for a unique function case.

Dynamic dispatch vs. Static overriding

Dynamic dispatch is the name for the language mechanism that chooses the implementation of a function based on the run-time class of an object. This constrasts with static overriding, which chooses the implementation of a function based on the compile-time types of that function's arguments.

For example, consider the following Java code. The Shape and Rectangle class both overload the intersect method: one version accepts a Shape argument, while the other accepts a Rectangle argument. Java performs dynamic dispatch based on the run-time class of the receiver, but uses static overloading to choose functions based on the compile-time types of the arguments. In Java, the overloading rule is applied first, followed by the dynamic dispatch rule.

```
class Shape {
  boolean intersect(Shape s) { ... } // impl #1
  boolean intersect(Rectangle r) { ... } // impl #2
}
class Rectangle extends Shape {
  boolean intersect(Shape s) { ... } // impl #3
  boolean intersect(Rectangle r) { ... } // impl #4
}
Shape s1 = new Shape(...);
Shape s2 = new Rectangle(...);
Rectangle r = new Rectangle(...);
```

Consider what happens if we call `s2.intersect(r)`. In this case, since r has static type Rectangle, the overloaded function that takes a Rectangle argument will be chosen, and dynamic dispatch will select implementaiton #4. However, if we call `r.intersect(s2)`, the system will choose the overloaded function that takes a Shape as an argument, since s2 has static type Shape (even though it's "really" a Rectangle underneath). Then dynamic dispatch will choose implementation #3, based on the actual object inside r.

As the example above illustrates, static overloading very rarely gives you what you want. Ideally, you would get dynamic dispatch semantics for not just the receiver but all of the arguments as well. That way, the method implementation would be selected based on real classes of the receiver and the arguments, not just what is statically visible.

A multi-method language is an object-oriented language where function dispatch can depend on all of the arguments to the function, not just the receiver of the message ("this" in Java or C++). EML is a multi-method language, because function cases can specify dispatch on every argument in the tuple that is passed to the function. For example, if we re-write the code above in EML, we get the right semantics:

```
class Shape of {...}
class Rectangle extends Shape of {...}
fun intersect : Shape * Shape -> boolean
extend fun intersect (s1 as Shape, s2 as Shape) =
               (* impl #1 *)
extend fun intersect (s as Shape, r as Rectangle) =
               (* impl #2 *)
extend fun intersect (r as Rectangle, s2 as Shape) =
               (* impl #3 *)
extend fun intersect (r1 as Rectangle, r2 as Rectangle) =
               (* impl #4 *)
val s1 : Shape = Shape{...}
val s2 : Shape = Rectangle{...}
val r : Rectangle = Rectangle{...}
```

Now, implementation #4 of intersect will be invoked no matter if we call `intersect(s2,r)` or `intersect(r,s2)`.

Modular Implementation Side Typechecking.

The implementation side typechecking rule defined above is global, which is undesirable since it means that we can't ensure that two modules that typecheck in isolation will also typecheck when they are combined. It is preferable to have a modular typechecking algorithm. We say a typechecking algorithm is *modular* if it can typecheck a module by looking at only the implementation of that module and the signatures of the modules it statically depends on. We say that a module M statically depends on signature S if M refers to some member of S, or if M depends on some other signature S' and S' refers to a member of S. In EML, we broaden the definition of signature slightly to include the list of cases defined for each function in a module.

The following table summarizes the rules that Java, ML, and EML use to modularly check for the absence of message not understood and message

ambiguous errors. The table shows that EML's strategy is a combination of the strategies of ML and of Java.

Implementation Side Typechecking	Message Not Understood	Message Ambiguous
Java	Concrete classes must define inherited abstract methods, and all argument types except the receiver must match the signature	Each function cases must be defined in the receiver class body
ML	All cases must be covered in a single local declaration (compiler warning if cases are not exhaustive)	All cases are declared together, and the order resolves any ambiguity
EML	External functions must provide a default case where function is declared. For internal functions, a local default case must be defined whenever a new concrete subclass of an abstract class is defined.	All function cases must be declared either in the same module where the function is declared, or in the module where the receiver is declared.

With the restrictions above applied to the EML language, the global typechecking algorithm can be run one module at a time, taking the $Decls$ set to be just the locally-declared declarations as well as the declarations in module signatures that the current module statically depends on. The authors of EML have proved that these restrictions are sufficient to guarantee that when the typechecking algorithm is run on several modules in isolation, global typechecking will succeed when these modules are linked. The proof is beyond the scope of this course, however.

Using Inheritance Correctly.

Consider the code below.

```
class Engine of { horsepower:int }
class Automobile extends Engine of { mileage:int }
class Airplane extends Engine of { prop:Propeller }
```

This code represents a broken use of inheritance. Essentially, it mixes up the subtyping relationship with the containment relationship. An Automobile has an Engine, and an Airplane has an Engine as well. However, "has a" relationships should be expressed with containment. For example, an Airplane has a Propeller, and this is properly expressed by giving the Airplane a prop field of type Propeller. Inheritance imposes a subtyping relationship, meaning that the declarations above state (incorrectly) that an Automobile is a Engine. This breaks down in an obvious way when you consider that not all operations which apply to engines also apply to Airplanes. For example, it makes sense to change the oil of an engine, but changing the oil of an airplane doesn't make sense in the same way. Using inheritance for a "has a" relationship also breaks down when we consider an Airplane that has more than one engine.

The general rule is: only use inheritance to represent an "is a" relationship between entities, never a "has a" relationship.