CSCI 2041: Object Systems

Chris Kauffman

Last Updated:
Fri Dec 7 09:00:45 CST 2018
Logistics

Reading

▶ Module Lazy on lazy evaluation
▶ Module Stream on streams
▶ OSM: Ch 3: Objects in OCaml

Goals

▶ Finish Lazy/Streams
▶ Define OO
▶ Objects and Classes in OCaml
▶ Dynamic Dispatch

Endgame

<table>
<thead>
<tr>
<th>Date</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wed 12/05</td>
<td>Lazy, Objects A5 Milestone</td>
</tr>
<tr>
<td>Fri 12/07</td>
<td>Object Systems</td>
</tr>
<tr>
<td>Mon 12/10</td>
<td>Optimization / Evals</td>
</tr>
<tr>
<td>Tue 12/11</td>
<td>Lab14: Review A5 Due</td>
</tr>
<tr>
<td>Wed 12/12</td>
<td>Last Lec: Review</td>
</tr>
<tr>
<td>Thu 12/13</td>
<td>Study Day</td>
</tr>
<tr>
<td>Mon 12/17</td>
<td>Final Exam</td>
</tr>
<tr>
<td>9:05am Sec 001</td>
<td>10:30am-12:30pm</td>
</tr>
<tr>
<td>1:25am Sec 010</td>
<td>1:30pm-3:30pm</td>
</tr>
</tbody>
</table>
Exercise: A Challenging Definition

- All of you should have previously taken a class on **object-oriented programming** (OOP) in some language.
- We are now 95% through a course on **functional programming** (FP) in OCaml.
- What’s the difference?
- Particularly, how would you distinguish what OOP has that FP does not?
- Draw from your experience in and be **rigorous**: ask questions like "Java has X, does OCaml have that?"
- Ultimately, define **object-oriented programming** to distinguish it from functional programming.
Answers: A Challenging Definition

- Disclaimer: this is a philosophical question so **there isn’t a strictly correct answer**
- Important to recognize things that are **not unique to OOP** that sensible FP languages have such as
  - Coupled functions and data (module with type and associated operations)
  - Strong data typing discipline
  - Rich data types (records, variants, tuples, arrays, lists)
  - Information hiding (signatures, lexical scope)
  - Interfaces (modules, functors, signatures)
  - "Constructors" (functions that create data)
  - Type neutral algorithms/data structs (polymorphism, functors)
  - State and Mutation (refs, mutable fields)
- What remains in OOP that we haven’t seen in OCaml?
  - Objects/Classes - not particularly useful on their own but...
Qualities of OOP

- An object/class system usually allows inheritance, sharing of code and structure which allows variation and specialization.
- Allows a codebase to be extended with new classes later and remain compatible with previous code.
- Also implies dynamic dispatch on method invocation: select the appropriate function to run based on the type of data passed to the function.
- So far we have not seen this capability in OCaml.
  - Possible to arrange code/structure sharing with Functors but not easy to vary individual pieces like a single module function.
  - Functions have static input types, can’t change behavior based on input type.
- For this, it is time to put the O in OCaml.
Classes and Objects in OCaml

- OCaml was originally Caml, then had a Class/Object System added to it to make it Objective Caml, shortened to OCaml
- Examine animals.ml for syntax around classes and objects
- Reminiscent of object systems in other languages though OCaml does not require objects to belong to a class\(^1\)
- Like Java’s abstract classes, can declare virtual classes leaving some methods unspecified
  - Cannot make new instances of virtual classes
- Subclasses inherit methods and fields from a base class but can override methods to behave differently
  - Subclass must implement virtual methods to be concrete or remain virtual

---

\(^1\)Examples of declaring objects without a classes are in OSM Ch 3.2: Immediate Objects. Java can do this in some circumstances as well.
class virtual animal = (* virtual: some methods un-implemented *)
  object(this) (* refer to object via 'this' *)
  method virtual id : unit -> string (* method not implemented *)
  method say () = (* implemented method *)
      printf "I'm a %s\n" (this#id ())
  end;;

class fish = (* another class *)
  object(me) (* refer to object via "me" *)
  inherit animal (* subclass of animal *)
  method id () = "fish" (* id method specified *)
  method say () = (* say method inherited *)
    printf "quack\n"
  end;;

class duck = object (* another class *)
  inherit animal (* subclass of animal *)
  method id () = "duck" (* override both methods *)
  method say () =
      printf "quack\n"
  end;;

class mascot = object (* subclass of duck *)
  inherit duck (* inherits id method *)
  method say () = (* overrides say method *)
      printf "Aflack!\n"
  end;;
Exercise: Single Dynamic Dispatch

```ocaml
let _ =
  let animals = [|(* main function *)
    ((new fish) :> animal); (* array of animals *)
    ((new duck) :> animal); (* "upcast" required to satisfy *)
    ((new mascot) :> animal); (* type checker: all array elems *)
    ((new fox) :> animal); (* elements of list are thus same *)
  |]

in
let len = Array.length animals in (* iterate over animals *)
for i=0 to len-1 do (* invoke id() method *)
  let a = animals.(i) in (* invoke say() method *)
  printf "The %s says: " (a#id ()); (* iterate over animals *)
  a#say ();
  done;
;;
```

Output is shown to the right

Why different for each animal?

How does this work at runtime?

OUTPUT:
> ocamlc animals.ml
> a.out
The fish says: I’m a fish
The duck says: quack
The duck says: Aflack!
The fox says:
Ring-ding-ding-ding-dingeringeding!
Gering-ding-ding-ding-dingeringeding!
Gering-ding-ding-ding-dingeringeding!
Answers: Single Dynamic Dispatch

- The output is different for each animal as each implements different versions of the `id()` and `say()` methods.
- At runtime, these methods dispatch to the most specific function most relevant to the class associated with the object.
- Dispatch involves a search process:
  - Determine type of object associated
  - Look for a function with method name in object's class
  - If not found, look in parent class
  - If not found, look in parent's parent class
  - etc.
- This search is handled at a low level by the runtime system which usually tries to optimize the process by remembering/caching what function to call for repeated invocations.

- Important trade-offs for function calls

<table>
<thead>
<tr>
<th>Call Type</th>
<th>Quality</th>
<th>Flexibility</th>
<th>Speed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-object Func Calls</td>
<td>Static</td>
<td>Less flexible</td>
<td>Constant Time</td>
</tr>
<tr>
<td>Method Dispatch</td>
<td>Dynamic</td>
<td>More flexible</td>
<td>Search Required</td>
</tr>
</tbody>
</table>
Single Dispatch Limits

- Most OOP languages perform Single Dynamic Dispatch on method invocations
- They **do not perform dynamic dispatch** in any other case
- In particular, don’t dispatch on function argument types which are determined at compile time, not runtime
  ```java
  public static void identify(Animal x) { // No dispatch
      System.out.println("I’m an animal");
  }
  public static void identify(Mouse x) { // No dispatch
      System.out.println("I’m a mouse");
  }
  ...
  Animal a = new Mouse();
  identify(a); // I’m an animal
  
  Further examples in SingleDispatch.java and DoubleDispatch.java
  ```
OOP Defined . . . right?

- **Methods** define a family of functions
- An object that implements a method will have a function of that name specific to its implementation which is used at runtime
- Early OOP languages like Smalltalk treated function calls as "messages" to object which would perform appropriate actions or respond "don’t know how to do that"

> "Actually I made up the term "object-oriented", and I can tell you I did not have C++ in mind." – Alan Kay

- OOP has a long history of such dynamic behavior and **dynamic dispatch** is at the center of it: pick the function appropriate to the object type
- So OOP must mean dynamic dispatch. Right. Right?  

Actually . . .

---

2 Co-author of the Smalltalk programming language (an early OOPL), Co-inventor of the Graphical User Interface
Dispatch as a Language Feature

- Java, Python, C++, OCaml feature Single Dynamic Dispatch: select a specific function based on the object type
- **Multiple Dynamic Dispatch** selects an appropriate function based on types of all arguments at runtime.
- MDD is an extremely useful feature for solving *interactions between types* of data such as below.

# Julia programming language uses multiple dispatch on types of all arguments to functions. New versions of collide for new types can be added later.

```plaintext
collide(x::Asteroid, y::Asteroid) = # asteroid hits asteroid
... 
collide(x::Asteroid, y::Spaceship) = # asteroid hits spaceship
... 
collide(x::Spaceship, y::Asteroid) = # spaceship hits asteroid
... 
collide(x::Spaceship, y::Spaceship) = # spaceship hits spaceship
... 
```

- Look for MDD/Multimethods in **Clojure, Julia, Racket, Common Lisp**, and others that are mostly **not** object-oriented
So what distinguishes OOP from FP?

- OOP is best understood as a mindset: model problem as classes of related, interacting objects
- In contrast, FP focuses on data types and the functions that operate on them
- Select a style that suits the problem at hand acknowledging the basic trade-offs of each

- OOP: class-centric
  - Each class implements its own methods
  - Adding a class is easy: define all its methods
  - Adding a method may require editing all classes to include the new method

- FP: function-centric
  - Each function defines behavior for all types
  - Adding a function is easy: define behavior for all types
  - Adding a type may require editing all functions to include the new type
The Connoisseur and the Carpenter

*If all you have is a hammer, everything looks like a nail.*
–Abraham Maslow

- A connoisseur will turn their nose up at one language or another for their off-putting qualities
- In contrast, carpenters use saws to cut, hammers to pound, drills to make holes, never viewing one tool as universally better, just better suited to different tasks
- **Good programmers are like carpenters** who can select an appropriate tool to get a job done easier, faster, and more robustly (leaving more time for Youtube)
- Programming Languages and Features are tools to address problems that arise in writing code
- Hopefully this course has given you an appreciation of FP as a valid and useful tool, worthy of inclusion in your box